

Introducción a la programación y al análisis de datos
con Python

Introducción

Índice

Esquema	3
Ideas clave	4
1.1. Introducción y objetivos	4
1.2. ¿Qué es Python?	5
1.3. Instalación	9
1.4. Herramientas	18

Introducción a Python		
¿Qué es Python?	Instalación	Entornos de desarrollo
Historia de Python	<div>Distribución básica de Python</div> <ul style="list-style-type: none">DescargaInstalaciónInstalación de nuevos módulos	Modo interactivo
Ventajas de Python		IPython
Versiones de Python		Editores de texto plano
	<div>Distribución Anaconda</div> <ul style="list-style-type: none">DescargaInstalaciónInstalación de nuevos módulos	Entornos de desarrollo avanzados
		Jupyter Notebook
		<ul style="list-style-type: none">Navegador de archivosVista del <i>notebook</i>

1.1. Introducción y objetivos

Python se ha convertido en uno de los lenguajes de programación más populares debido a su potencia y su facilidad para aprenderlo. En este tema pondremos en contexto qué es Python y qué características han hecho que este lenguaje sea uno de los más utilizados, sobre todo en campos de inteligencia artificial o análisis de datos.

Además, en este tema prepararemos nuestros equipos para poder desarrollar en Python. Para ello, explicaremos cómo instalar las dos distribuciones más populares dentro de Python y cómo instalar nuevos módulos en estas distribuciones para aumentar el número de funcionalidades. Por último, describiremos diferentes herramientas disponibles para desarrollar en este lenguaje y nos centraremos en Jupyter Notebook, que será el entorno de desarrollo que usaremos a lo largo del curso. Los objetivos que trataremos son:

- ▶ Contextualizar Python desde su historia y sus características.
- ▶ Comprender el problema de las versiones que ha existido hasta este año.
- ▶ Conocer los pasos para instalar Python en nuestro equipo.
- ▶ Conocer las distintas herramientas existentes para programar en Python.
- ▶ Comprender el entorno de desarrollo de Jupyter Notebook.

1.2. ¿Qué es Python?

Python es un lenguaje de propósito general que en los últimos años se ha ido haciendo cada vez más popular en áreas como *data science* o la inteligencia artificial.

Contexto de Python

Para entender un poco de dónde viene y por qué se ha vuelto tan popular, vamos a repasar su historia y las características más importantes.

Historia

Python fue creado en 1989 por Guido van Rossum. Guido empezó a implementar este lenguaje como pasatiempo con el objetivo de que fuera fácil de usar y aprender, pero, a la vez, que fuese un lenguaje potente.



Figura 1. Guido van Rossum, creador del lenguaje de programación Python. Fuente: <http://perlcgi-book.com/tag/guido-van-rossum/>

Van Rossum le dio el nombre de «Python» en homenaje al grupo Monty Python del que era fan. Aunque su desarrollo empezó en 1989, la primera versión no se publicaría hasta principios de 1991. Como hemos dicho, el objetivo principal de este

lenguaje era que fuese fácil de entender, es decir, que el código que se escribiese con Python fuese más legible que con otros lenguajes de la época como C++ o Java.

Para lograr este objetivo, el desarrollador Tim Peters creó el Zen de Python, que define un conjunto de reglas que representan la filosofía de Python. Todos los usuarios pueden acceder a este conjunto de reglas a través de un *easter egg* que se introdujo en Python. Para verlo, solo tenemos que ejecutar la instrucción `import this` en una terminal con Python.

Siguiendo la filosofía propuesta en el Zen de Python, se creó una guía de estilo que se encuentra descrita en el *Python Enhancement Proposal*, abreviado como PEP, versión 8. Esta guía contiene las reglas de estilo que se aplicaron a la hora de desarrollar Python para que se sigan en el desarrollo de nuevas aplicaciones.

PEP 8 -- Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Figura 2. Información de la guía de estilos de Python PEP8. Fuente: <https://www.python.org/dev/peps/pep-0008/>

Toda esta filosofía está muy bien, pero Python no es tan popular solo por sus reglas de estilo y su filosofía. En la siguiente sección veremos las ventajas más importantes de este lenguaje de programación.

Ventajas de Python

Python tiene varias propiedades que lo han convertido en un lenguaje muy potente y fácil de aprender. Estas propiedades son las siguientes:

- ▶ **Tipado dinámico:** Python no necesita que definamos el tipo de las variables cuando las inicializamos como pasa, por ejemplo, en Java o C. Cuando inicializamos una variable Python le asigna el tipo del valor que le estamos asignando. Incluso, durante la ejecución, una misma variable podría contener valores con distintos tipos. Esta propiedad hace que sea más sencillo aprender a usarlo, aunque hace que sea más difícil detectar errores asociados con los tipos de datos.
- ▶ **Lenguaje multiparadigma:** Python permite aplicar diferentes paradigmas de programación como son la programación orientada a objetos, como Java o C++, programación imperativa, como C, o programación funcional, como Haskell.
- ▶ **Interpretado/*scripts*:** otra ventaja es que podemos ejecutar Python de forma interpretada o usando *scripts*. Es decir, puedo abrir una consola de Python y escribir y ejecutar las instrucciones una a una o, por otro lado, puedo crear un fichero que almacene todo el programa.
- ▶ **Extensible:** por último, Python cuenta con una gran cantidad de módulos y librerías que podemos instalar para incluir nuevas funcionalidades. Sin embargo, como Python esta implementado usando C++, podemos crear nuevos módulos en C++ e incluirlo en Python haciendo que el lenguaje sea extensible a nuevos módulos.

Versiones en Python

Uno de los problemas que tenía una persona que se inicializaba en Python era saber qué versión tenía que instalar, ya que hasta finales de diciembre de 2019 existían dos versiones activas.

Versiones de Python

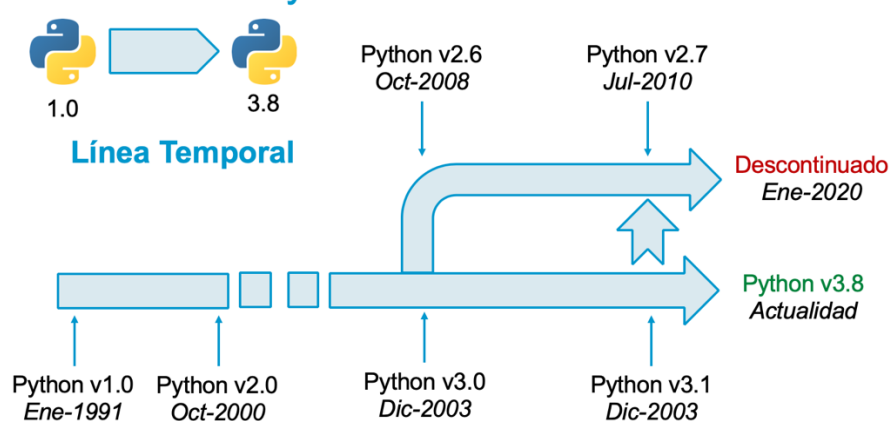


Figura 3. Línea temporal de las versiones de Python.

Como explicamos al principio, la primera versión de Python se publicó en 1991. Desde entonces se han publicado varias versiones que han seguido siendo más o menos retrocompatibles. Es decir, podía usar esas versiones en programas que había creado en versiones anteriores.

Sin embargo, a finales de 2008 se iba a publicar la versión 3.0. Esta nueva versión era un cambio radical con respecto a las predecesoras y esto hacía que no fuera compatible con las versiones anteriores de Python. Por este motivo, se publicó la versión 2.6 como soporte para los desarrollos de la versión 2. En la versión 2.6 se incluyeron nuevas funcionalidades de la versión 3, pero adaptadas a la versión 2. Desde ese momento Python contaba con 2 versiones y las novedades las tenían que duplicar en ambas. Por ejemplo, en 2010 publicaron la versión 3.1 y la 2.7 que incluía las nuevas funcionalidades, pero adaptadas a la versión 2.

Sin embargo, desde el equipo de Python siempre han explicado que las versiones 2.6 o 2.7 eran un parche y que no iban a ser versiones funcionales en un futuro. Este hecho se hizo oficial en enero de 2020 cuando se decidió que la versión 2.7 quedaba descontinuada y que a partir de entonces solo se publicarían nuevas funcionalidades para la versión 3.

En este curso trabajaremos con la versión 3.8 de Python, ya que en el momento de publicar este vídeo es la versión más reciente de Python dentro de la rama que publicará nuevas funcionalidades.



Vídeo 1. ¿Qué es Python?

Accede al vídeo a través del aula virtual

1.3. Instalación

Distribución básica Python

La primera opción que podemos instalar es una distribución básica de Python. Esta distribución solo incluye los módulos principales que hay en Python. A continuación, veremos los pasos para instalarlo.

Descargar Python

La primera opción de instalación es únicamente los módulos principales del lenguaje. En esta opción tendremos que instalar nuevos módulos si queremos ampliar las funcionalidades de Python. Para poder descargarnos esta versión, deberemos acceder a la sección de descargas de la página principal de Python.



Figura 4. Página principal de descargas de Python. Fuente: <https://www.python.org/downloads/>

En esta página tenemos una opción para instalar la última versión estable de Python. En nuestro caso, es la versión 3.8. Para descargarlo, hacemos clic en el botón *download Python 3.X* y automáticamente empezará la descarga del paquete de instalación.

Instalación Python

Una vez que tenemos el paquete de instalación descargado, podemos iniciar el proceso de instalación. Para ello, ejecutaremos el fichero *python-3.X.exe* que hemos ejecutado y, a continuación, se nos abrirá el asistente de configuración.

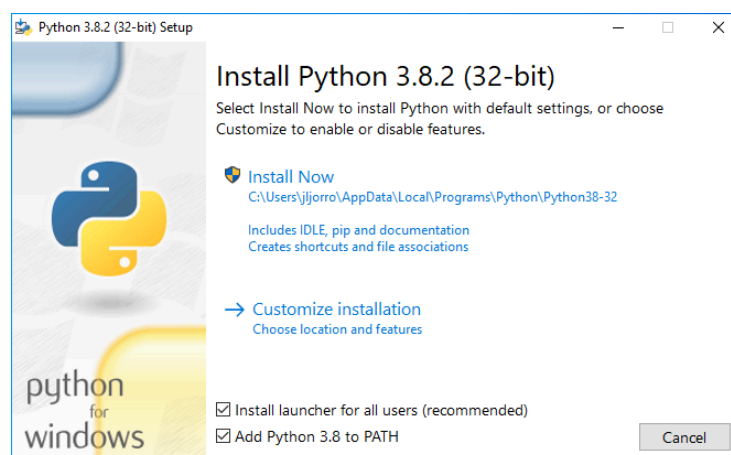


Figura 5. Asistente de configuración de la instalación de Python.

La primera ventana del asistente nos muestra dos formas para instalar Python. La primera de ellas, *install now*, nos permite instalar Python con la configuración por defecto. En cambio, *customize installation* permite cambiar algunos parámetros como, por ejemplo, dónde vamos a instalar Python. Por último, tenemos dos opciones que tenemos que comprobar que están marcadas para, en primer lugar, instalar Python a todos los usuarios de un equipo y, en segundo lugar, almacenar Python en el PATH del sistema operativo. Para nuestra instalación elegiremos la opción *install now*.

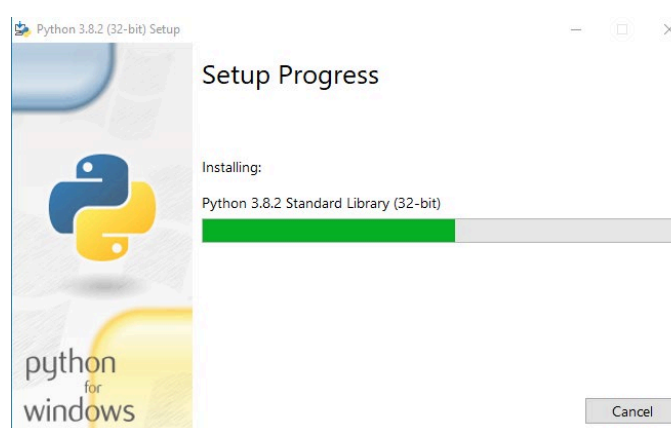


Figura 6. Progreso de la instalación de Python.

A continuación, aparecerá una ventana como la Figura 6, en la que se mostrará el progreso de la instalación de Python. Pasados unos minutos, el asistente nos informará de que la instalación ha terminado con éxito y podremos cerrar el asistente con el botón *close*.

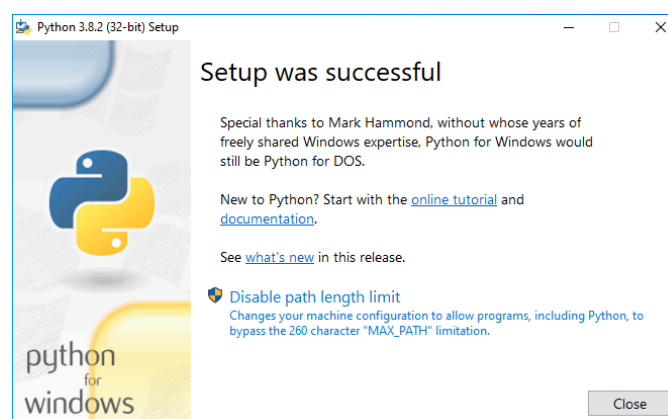
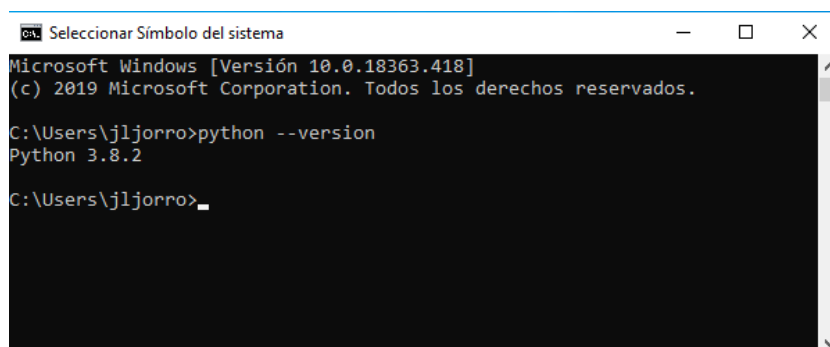


Figura 7. Confirmación de que la instalación se realizó correctamente.

También podemos comprobar que la instalación se realizó correctamente. Una forma de comprobarlo es preguntando al sistema por la versión que tenemos instalada de Python. Para ello abrimos el símbolo del sistema y ejecutamos la instrucción `python --version`. Esto nos debería mostrar la versión de Python que hemos instalado.



```
Microsoft Windows [Versión 10.0.18363.418]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\jljorro>python --version
Python 3.8.2

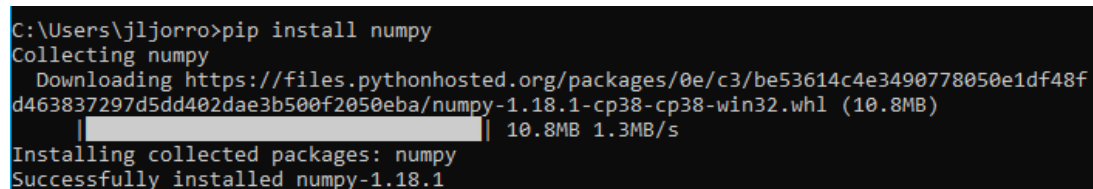
C:\Users\jljorro>
```

Figura 8. Comprobación de la instalación desde el símbolo del sistema.

Instalación de nuevos módulos

La versión que hemos instalado solo contiene los módulos básicos de Python. Por este motivo suele ser necesario instalar los nuevos módulos que queremos incluir en nuestros programas. Para instalar nuevos módulos, Python incluye el paquete de instalación para Python (pip).

Si queremos instalar un nuevo módulo en Python, tenemos que ejecutar la instrucción `pip install nombre_modulo`. Por ejemplo, si queremos instalar el módulo *numpy*, abriremos el símbolo del sistema y ejecutaremos la instrucción de instalación.



```
C:\Users\jljorro>pip install numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/0e/c3/be53614c4e3490778050e1df48fd463837297d5dd402dae3b500f2050eba/numpy-1.18.1-cp38-cp38-win32.whl (10.8MB)
    |#####| 10.8MB 1.3MB/s
Installing collected packages: numpy
Successfully installed numpy-1.18.1
```

Figura 9. Comprobación de la instalación desde el símbolo del sistema.

De esta forma podremos instalar todos los módulos que se puedan necesitar en el futuro. Sin embargo, hay otra distribución que tiene algunos módulos preinstalados, sobre todo aquellos asociados a la *data science*. En el siguiente capítulo veremos cómo instalar esta distribución.

Distribución Anaconda

Otra distribución con la que podemos instalar Python es Anaconda. Esta distribución, aparte de instalar la versión básica de Python, incluye otros módulos importantes dentro del análisis de datos, como son *numpy* o *pandas*. Esta es la distribución que usaremos en el curso.

Descargar Anaconda

Otra distribución de Python es la que nos proporciona Anaconda. Esta distribución nos instala los módulos principales de Python y otros módulos importantes en el análisis de datos como son *numpy* y *pandas*. Esta es la distribución que usaremos durante el curso. Para descargarnos Anaconda, nos dirigimos a la sección de descargas de su página y seleccionamos la versión 3.7.

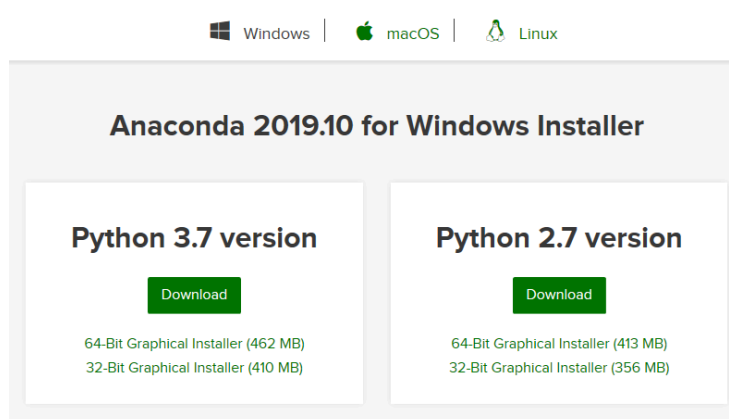


Figura 10. Página principal para descargar distintas versiones de Anaconda. Fuente: <https://www.anaconda.com/products/individual#download-section>

Una vez descargada la versión de Anaconda, procederemos a su instalación en nuestro equipo.

Instalación Anaconda

Para iniciar la instalación de Anaconda en nuestro equipo, ejecutamos el fichero *Anaconda3-XXX.exe*. Hecho esto, se nos abrirá el asistente de instalación que nos guiará en los pasos de configuración de Anaconda.

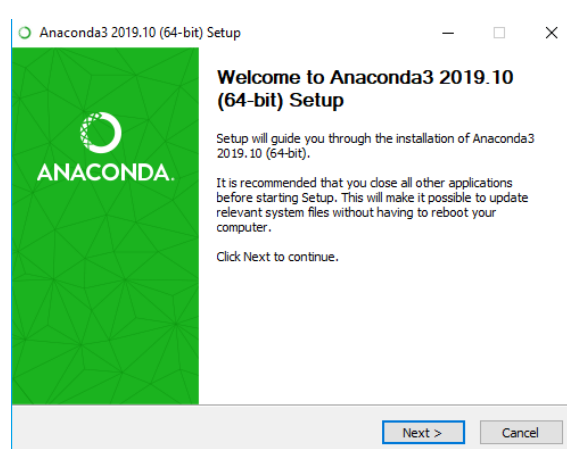


Figura 11. Ventana bienvenida instalación Anaconda.

Seleccionamos el botón *Next >* y nos mostrará el acuerdo de licencia del *software*. Para continuar con la instalación, es necesario aceptar los términos de la licencia con el botón *I Agree*.

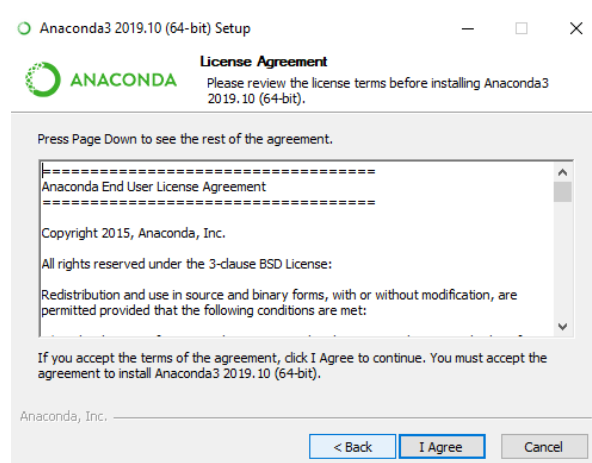


Figura 12. Acuerdo de licencia de Anaconda.

En el siguiente paso, el asistente nos preguntará el tipo de instalación que queremos hacer. La primera opción es hacer una instalación únicamente para el usuario que se está ejecutando en ese momento. La segunda opción permite instalar Anaconda a todos los usuarios que comparten un mismo equipo, aunque para ello necesita permisos de administración. Seleccionamos una de las dos opciones y continuamos con el botón *Next >*.

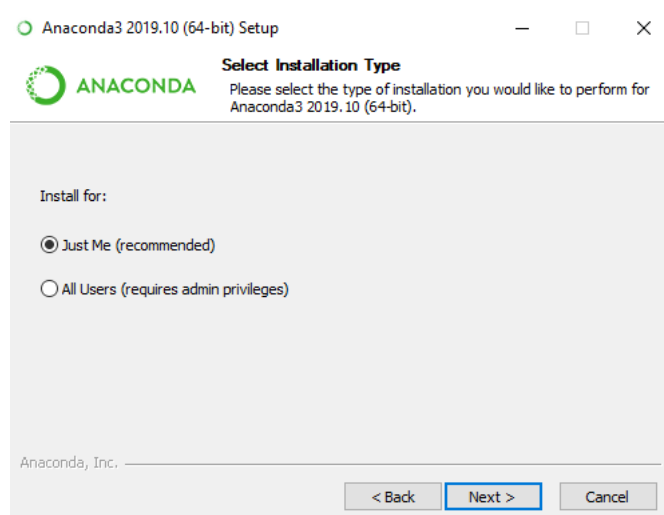


Figura 13. Selección del tipo de instalación de Anaconda.

Una vez decidido qué usuarios pueden acceder a Anaconda, toca elegir en qué carpeta deseamos que se instale el *software*. En nuestro caso dejaremos la carpeta que viene por defecto. En este paso hay que prestar atención al espacio libre que tenemos en el sistema, ya que Anaconda ocupa 2,9 GB. Si hay espacio suficiente y hemos elegido la carpeta, continuamos la instalación pulsando *Next >*.

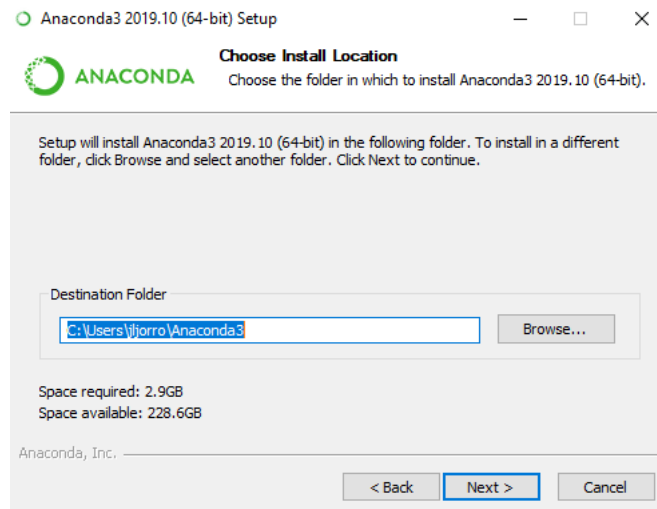


Figura 14. Selección de la localización de instalación de Anaconda.

El siguiente paso es muy importante. Tenemos que asegurarnos de que las dos opciones avanzadas que se nos muestran están seleccionadas. La primera de ellas incluye Anaconda en el PATH del sistema. Esto nos permitirá acceder a Anaconda desde el símbolo del sistema. La segunda opción hace que la versión de Python instalada en Anaconda sea la versión por defecto del sistema, en nuestro caso la versión 3.7. Una vez seleccionadas ambas opciones, continuamos la instalación con el botón *Install*.

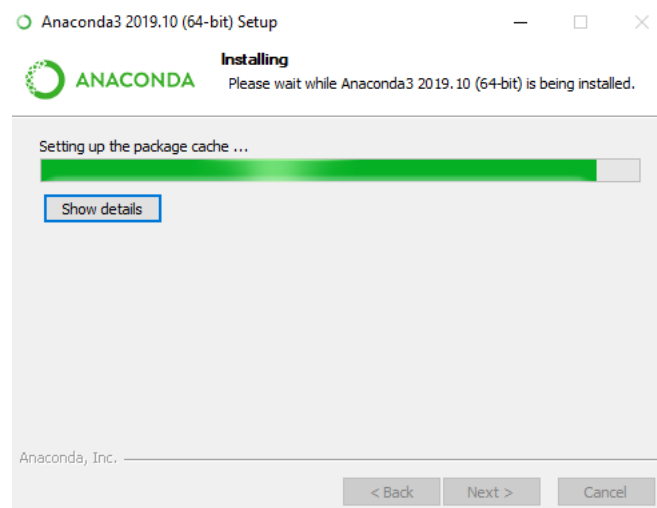


Figura 15. Progreso de instalación de Anaconda.

En ese momento, el asistente nos mostrará una barra de progreso de la instalación. Pasado un tiempo, el sistema nos indicará que la instalación se ha completado. En ese momento pulsamos el botón *Next >* para finalizar la instalación. En las ventanas siguientes nos mostrarán algunos mensajes de editores y opciones que podemos utilizar con Anaconda. Seguiremos dando al botón *Next >* hasta llegar al último paso donde pulsaremos el botón *Finish*.

Para comprobar que la instalación se ha realizado correctamente, nos dirigiremos al botón de inicio de Windows y entre los programas tendremos una carpeta con el nombre «Anaconda3».

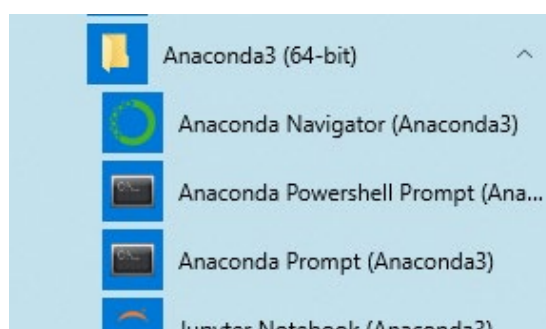


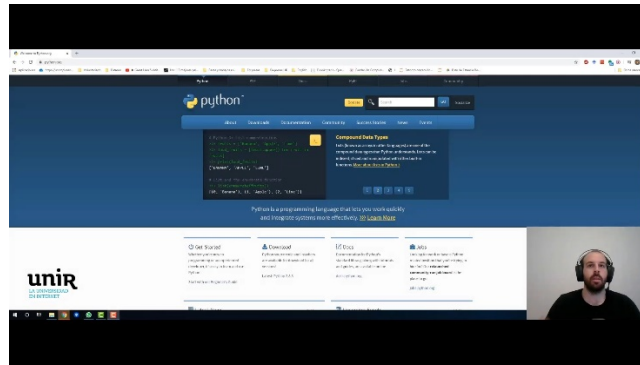
Figura 16. Localización de Anaconda en el menú de inicio de Windows.

Instalación de nuevos módulos

Aunque la distribución de Anaconda incluye nuevas funcionalidades a los módulos básicos de Python, no están incluidos todos los posibles. Por este motivo también puede ser necesario instalar nuevos módulos en nuestra distribución. Para instalar estos nuevos módulos, podemos utilizar la opción de instalación que tenemos desde el símbolo del sistema: `conda install nombre_módulo`. Por ejemplo, vamos a instalar el módulo *scikit-learn* en nuestro sistema.

```
C:\Users\jljorro>conda install scikit-learn
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

Figura 17. Comprobación de la instalación desde el Símbolo del sistema.



Vídeo 2. Instalación de Python.

Accede al vídeo a través del aula virtual

1.4. Herramientas

Entornos de desarrollo

Como ya hemos explicado, Python es un lenguaje que podemos ejecutarlo de dos formas principalmente: usando el intérprete o usando *scripts*. En este apartado veremos los diferentes entornos de desarrollo que encontramos para programar en Python. Además, explicaremos en detalle el funcionamiento de Jupyter Notebook, ya que será la herramienta con la que trabajemos en este curso.

Modo interactivo

Python es un lenguaje interpretado, es decir, Python es capaz de ir ejecutando las instrucciones según las vamos introduciendo. Por este motivo, la instalación de Python incluye el intérprete en el que podemos ejecutar instrucciones. Para iniciar este intérprete, solo tenemos que ejecutar la instrucción `python` en nuestra consola de comandos.

```
Python 3.8.2 (v3.8.2:7b3ab5921f, Feb 24 2020, 17:52:18)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figura 18. Intérprete de Python para la versión 3.8.

Usando este modo podemos conocer algunos elementos del lenguaje Python como sus clases o funciones. Además, podemos consultar la documentación del lenguaje desde el propio intérprete. Os animamos a que uséis el intérprete de Python para probar los conceptos básicos que vemos durante el curso. Para salir del intérprete solo debemos ejecutar el comando `exit()`.

IPython

Para mejorar el intérprete de Python, se puede instalar el paquete IPython (las instrucciones se encuentran en <https://ipython.org/install.html>). IPython añade más funcionalidades al intérprete de Python, como son el resaltado de errores, completado automático de variables o módulos a través del tabulador, etc. Una vez instalado, para iniciar este intérprete solo debemos ejecutar la instrucción `ipython`.

```
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: he
help() hex
```

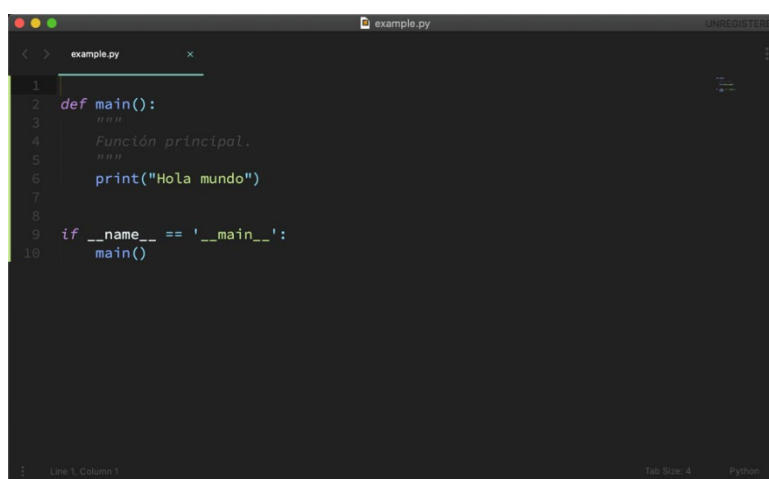
Figura 19. Intérprete IPython ejecutando el completado automático de nombres usando el tabulador.

Editores de texto plano

Las dos opciones anteriores nos permiten ejecutar pequeñas instrucciones en Python y poder consultar la documentación de objetos y módulos. Sin embargo, para crear programas más complejos es necesario escribir *scripts* que contienen más instrucciones o diferentes bloques de código. Una de las primeras opciones que se pueden utilizar para implementar estos *scripts* es la utilización de editores de texto

plano. Existen muchos tipos de editores de texto para todos los sistemas operativos, algunos ejemplos pueden ser:

- ▶ Nano o vim (sistemas UNIX).
- ▶ Bloc de notas de Windows.
- ▶ Sublime Text 3.
- ▶ Atom.
- ▶ Notepad++ (solo Windows).
- ▶ Visual Code.



```
1
2 def main():
3     """
4     Función principal.
5     """
6     print("Hola mundo")
7
8
9 if __name__ == '__main__':
10    main()
```

Figura 20. Ejemplo de *script* de Python implementado en el editor Sublime Text.

Entornos de desarrollo avanzados

Por último, existen diferentes entornos de desarrollo avanzados orientados a Python. Estos entornos están orientados a grandes proyectos en Python y se incluyen muchas más funcionalidades como son la gestión de repositorios. Algunos de estos entornos de desarrollo pueden ejecutar en un mismo proyecto *scripts* y *notebooks*, como es el caso de PyCharm. Los entornos de desarrollo más utilizados son:

- ▶ PyCharm.
- ▶ Eclipse PyDev.
- ▶ Spyder.

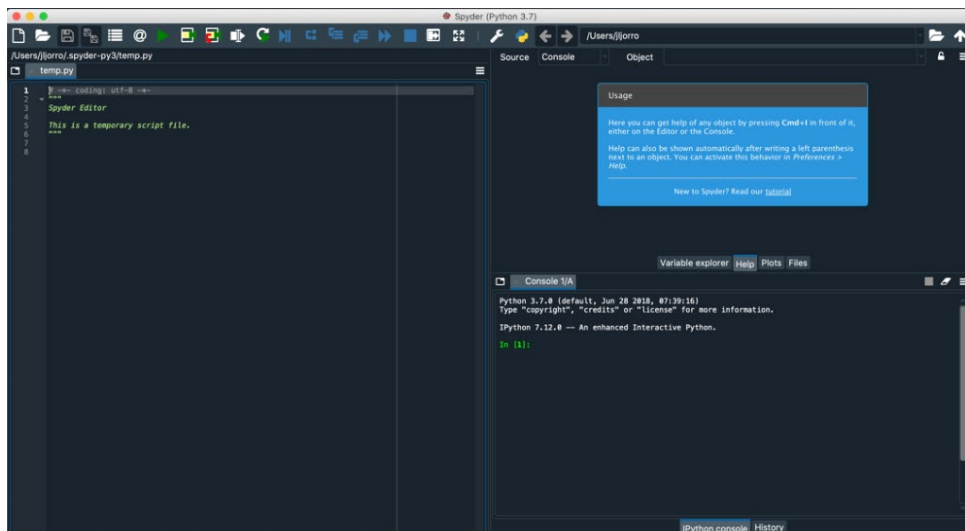


Figura 21. Ventana principal del entorno de desarrollo Spyder.

Jupyter Notebook

Jupyter Notebook es una aplicación web incluida en la distribución Anaconda. Esta aplicación web es una extensión de IPython, donde se añaden funcionalidades y se mejora la interfaz gráfica. La principal característica que tiene Jupyter Notebook es la creación de celdas con objetivos específicos. Estos objetivos específicos de cada celda pueden ser: ejecutar código de Python, incluir texto en *markdown* o visualizar gráficos. Es la aplicación más utilizada en el campo de la ciencia de datos.

Para abrir Jupyter Notebook en nuestro equipo, ejecutaremos la aplicación Anaconda Navigator. Esta aplicación nos mostrará distintas herramientas que podemos ejecutar. Desde ahí pulsamos en el botón *Launch* de Jupyter Notebook.

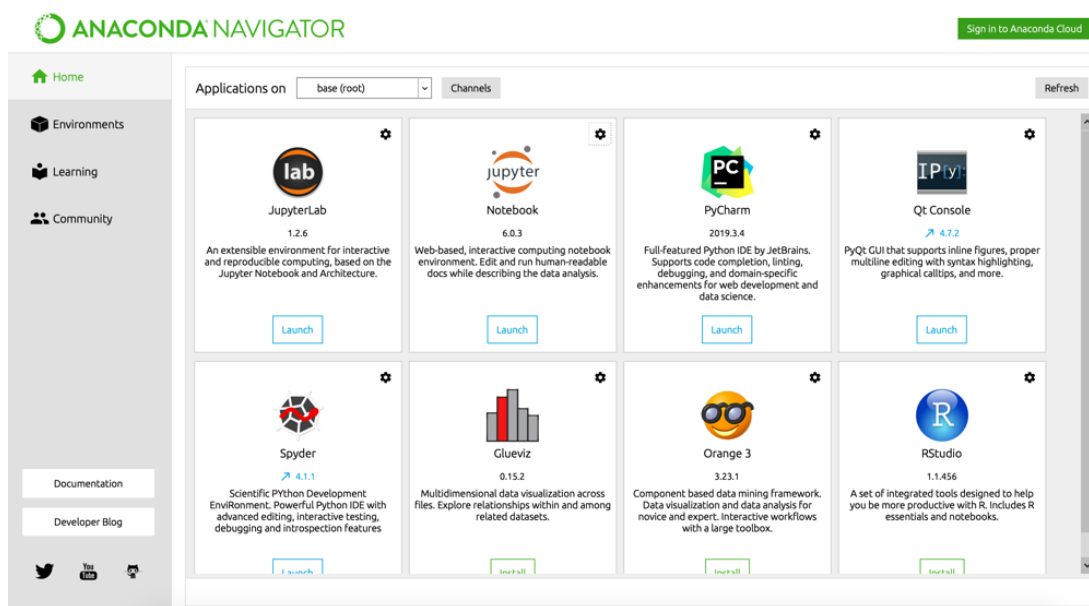


Figura 22. Vista de la ventana de aplicaciones de Anaconda Navigator.

Esto hará que nuestro navegador web predeterminado abra Jupyter Notebook como una aplicación web. La primera ventana que veremos será el navegador de archivos de Jupyter que explicaremos a continuación.

Navegador de archivos

La primera ventana que nos muestra Jupyter Notebook es el navegador de archivos. A través de esta ventana podemos movernos en nuestro sistema de ficheros, crear nuevas carpetas o archivos y renombrar ficheros.



Figura 23. Vista del navegador de archivos de Jupyter Notebook.

En primer lugar, para navegar entre las carpetas, solo tendremos que hacer clic en el nombre de la carpeta a la que queremos acceder. Si lo que queremos es volver a la carpeta superior, solo debemos hacer clic en la carpeta que tiene como nombre dos puntos (..).

Para crear un nuevo fichero, pulsamos sobre el botón *New*. Una vez hecho esto, se nos desplegarán varias opciones para crear un fichero: *notebook* (con la versión correspondiente de Python) u otro tipo de fichero como, por ejemplo, un fichero de texto o una carpeta.

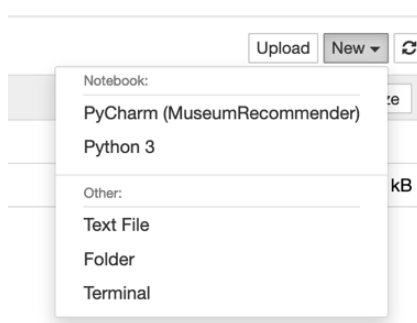


Figura 24. Menú con los tipos de fichero que se pueden crear.

Esto hará que se cree un fichero del tipo seleccionado en la carpeta en la que nos encontremos en ese momento. Vamos a crear un fichero de tipo *notebook* y, a continuación, explicaremos la ventana que nos muestra Jupyter dentro de un *notebook*.

Vista del *notebook*

Cuando creamos un *notebook* o abrimos uno que ya existe veremos una pantalla similar a la que se muestra en la Figura 24. En la parte inferior veremos todas las celdas de nuestro *notebook*. Cada una de estas celdas pueden ser de los siguientes tipos:

- ▶ Código en Python.
- ▶ Texto con *markdown*.
- ▶ Formato *raw* en la que se muestra el texto con el formato de consola.

- Formato *heading* que crea una celda con formato título.

Además, existe otro tipo de celda que se da cuando una celda de código en Python devuelve un resultado como, por ejemplo, si utilizamos la instrucción `print`. Estas celdas se crearán de forma automática.

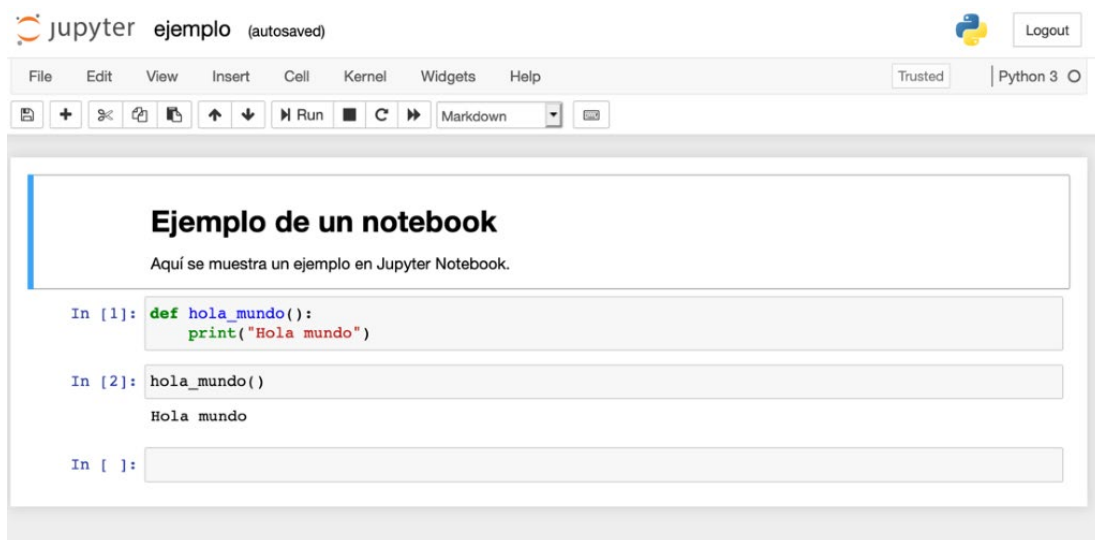


Figura 25. Ejemplo de un *notebook* en Jupyter.

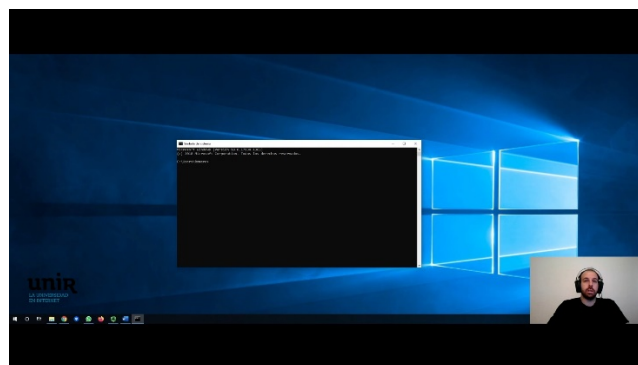
Justo encima de las celdas, tenemos un conjunto de botones que nos permiten interactuar con las celdas. A continuación, explicaremos cada uno de estos botones siguiendo el orden de izquierda a la derecha:

- **Guardar:** almacena en el fichero todas las celdas y guarda el estado de ejecución en el que se quedó el *notebook*.
- **Nueva celda:** crea una nueva celda inmediatamente después de la celda que tenemos seleccionada.
- **Cortar:** permite cortar una celda para pegarla en otro punto del *notebook*.
- **Copiar:** permite copiar una celda para poder pegarla en otro punto del *notebook*.
- **Pegar:** pegamos la celda que hemos copiado o cortado previamente inmediatamente después de la celda que tenemos seleccionada.
- **Bajar una celda:** desplaza la celda seleccionada una posición hacia abajo.
- **Subir una celda:** desplaza la celda seleccionada una posición hacia arriba.

- ▶ **Ejecutar celda:** ejecuta el contenido que hay en la celda seleccionada. Si esa celda es de tipo código, ejecutará las instrucciones y devolverá la salida en una celda de salida. Por otro lado, si la celda es de tipo texto, le asignará un formato HTML.
- ▶ **Stop:** para la ejecución del *kernel* de Python. Para poder seguir ejecutando nuevas celdas, es necesario reiniciar el *kernel* de Python.
- ▶ **Reiniciar *kernel*:** reinicia la ejecución del *kernel*, eliminando de la memoria toda la información del *notebook* que tuviese almacenada.
- ▶ **Reiniciar el *kernel* y ejecutar todas las celdas:** reinicia el *kernel* de Python como el botón anterior y, después, ejecuta todas las celdas del *notebook*.
- ▶ **Selección del tipo de celda:** permite seleccionar el tipo de celda que tenemos seleccionado. Los tipos son los que hemos descrito anteriormente.

En la parte superior se encuentra el menú que incluye muchas más funciones. A continuación, explicamos algunas de las funciones más útiles y dónde se encuentran:

- ▶ Crear un nuevo *notebook* [*File >> New Notebook*].
- ▶ Descargar *notebook* en otro formato (HTML, PDF...) [*File >> Downloads*].
- ▶ Cerrar el *notebook* y apagar el *kernel* [*File >> Close and Halt*].
- ▶ Insertar celdas encima o debajo de la celda seleccionada [*Insert*].
- ▶ Referencia de Python y librerías utilizadas en Jupyter [*Help*].



Vídeo 3. Herramientas para programar en Python.

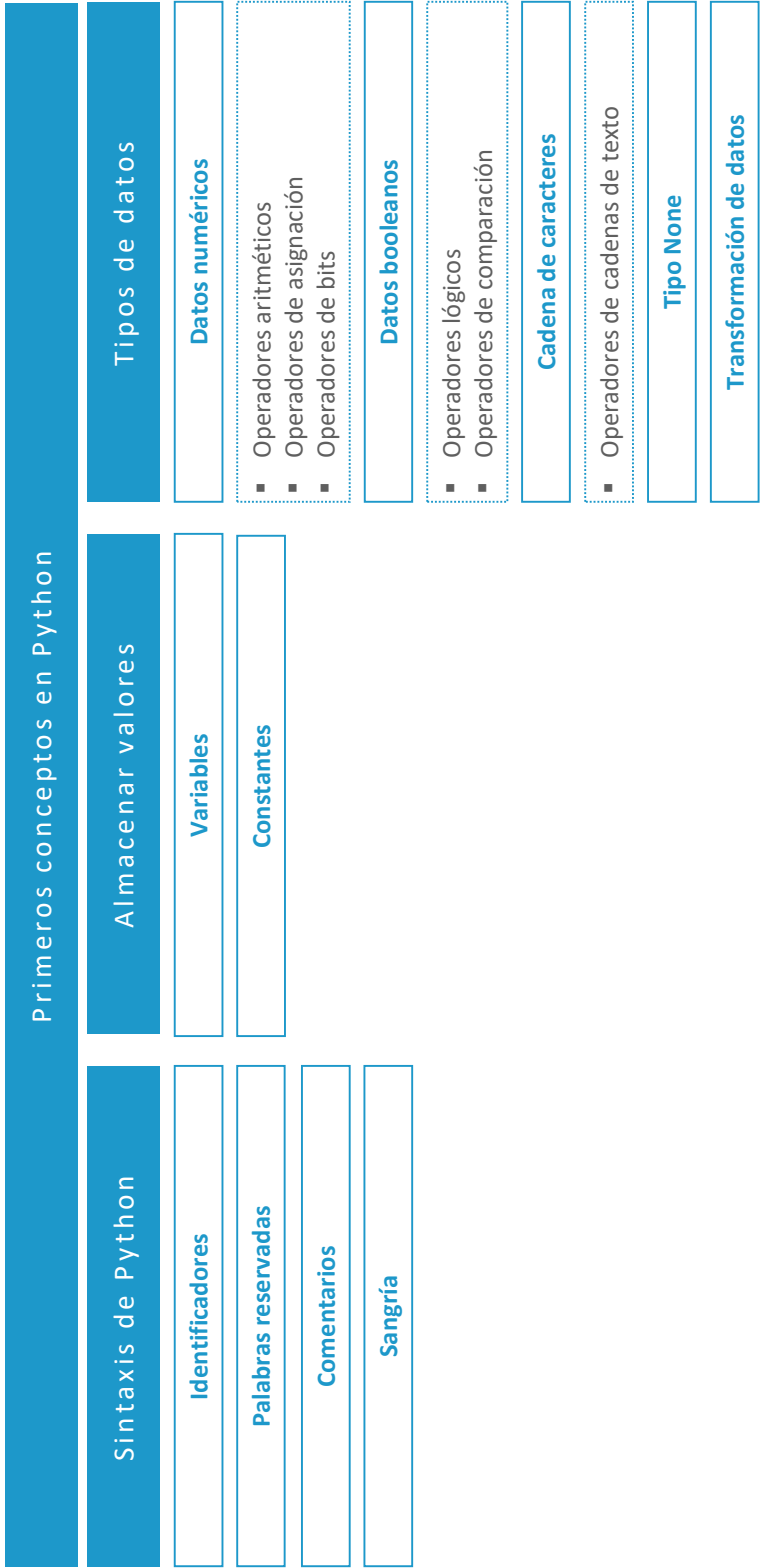
Accede al vídeo a través del aula virtual

Introducción a la programación y al análisis de datos
con Python

Primeros conceptos en Python

Índice

Esquema	3
Ideas clave	4
2.1. Introducción y objetivos	4
2.2. Sintaxis de Python	5
2.3. Almacenar valores	7
2.4. Tipos de datos y operadores	10



Esquema

2.1. Introducción y objetivos

En este tema trataremos los primeros conceptos que nos permitirán hacer nuestros programas en Python. En primer lugar, veremos algunas reglas de cómo se deben escribir los diferentes elementos en Python, es decir, su sintaxis. Esto es debido a que Python hace algunos cambios con respecto a otros lenguajes como Java y C para mejorar la legibilidad del código. A continuación, enumeraremos las dos formas que tenemos para almacenar valores en Python y poder usarlos durante la ejecución de nuestros programas. Por último, veremos los tipos de datos básicos que podemos utilizar en Python, algunos de los operadores de cada uno de estos tipos de datos y cómo cambiar el tipo de dato de una variable.

Al finalizar este tema, habrás alcanzado los siguientes objetivos para hacer tus primeros programas en Python:

- ▶ Conocer las sintaxis de Python para los identificadores, los comentarios y la sangría.
- ▶ Conocer cómo almacenar valores en variables y constantes.
- ▶ Comprender los tipos básicos de Python: numéricos, booleanos, cadenas de caracteres y el tipo None.
- ▶ Conocer los diferentes operadores que podemos aplicar a cada uno de los tipos de datos.
- ▶ Aprender cómo se puede cambiar el tipo de dato de las variables.

2.2. Sintaxis de Python

En este primer apartado, explicaremos algunas normas básicas en la sintaxis de Python para familiarizarnos con el lenguaje. Como explicamos al comienzo del curso, Python sigue una guía de estilo con el objetivo de que los programas sean fáciles de leer. Algunas de las normas que explicaremos aquí han sido creadas con el mismo objetivo. A continuación, detallaremos estas normas básicas.

Identificadores

Los identificadores son nombres que asignaremos a elementos, como funciones, variables, etc., para hacerles referencia más adelante en el código. Estos identificadores tienen que seguir las siguientes reglas:

- ▶ Pueden ser una combinación de números (0-9), letras mayúsculas (A-Z), letras minúsculas (a-z) y el símbolo de guion bajo (_).
- ▶ No pueden comenzar por un dígito.
- ▶ No pueden utilizarse símbolos especiales: @, !, #, etc.
- ▶ Pueden tener cualquier longitud.

Hay que decir que Python distingue las mayúsculas de las minúsculas. Esto significa que, si tuviéramos los identificadores `identificador` e `IDENTIFICADOR`, Python los consideraría identificadores diferentes. En el siguiente bloque veremos algunos ejemplos:

```
# Identificadores correctos.
variable = 1
otra_variable = 'Otra'
_mas_variables = [1, 3, 4]
Variable_4 = True
```

```
# Identificadores incorrectos.
@variable = 3
2_variable = 'Está mal.'
```

Palabras reservadas

Python, como todos los lenguajes de programación, tiene un conjunto de palabras reservadas que no se pueden utilizar como identificadores. Esta lista de palabras reservadas es la siguiente: and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield.

Comentarios

Los comentarios son partes del código que el intérprete de Python no ejecuta. Estos comentarios nos permiten escribir aclaraciones en el código para mejorar su interpretabilidad por parte de otros desarrolladores o para nosotros mismos en un futuro. En Python hay dos sintaxis diferentes para escribir los comentarios:

- **Comentarios de línea:** para aquellos comentarios que solo ocupen una línea se utiliza el símbolo # al principio de la misma.
- **Comentarios de bloque:** para los comentarios que ocupen más de una línea se utilizan triples comillas (simples o dobles) al principio y al final del comentario.

A continuación, se muestra un ejemplo de ambos tipos de comentarios:

```
# Ejemplo de comentario de línea.  
  
"""  
Ejemplo de comentario de bloque.  
Todas las líneas pertenecen al comentario.  
"""  
  
...  
Otro ejemplo de  
comentario de bloque.  
...
```

Sangría

Una de las diferencias que tiene Python con respecto a otros lenguajes, como C o Java, es que no utiliza llaves ({,}) para diferenciar bloques de código. En Python se utiliza la sangría de bloques, es decir, hacer una separación hacia la derecha del código que pertenece a un bloque. Esta sangría está determinada por 4 espacios según la guía de estilos PEP 8.

Esta forma de definir los bloques de código hace que el código sea más legible. Sin embargo, al ser obligatorio, puede llevar a errores de ejecución si no se ha hecho la sangría correctamente. A continuación, vemos un ejemplo de cómo funciona la sangría dentro de un bloque *if*.

```
# Ejemplo de una sangría:
variable = 1

if (variable == 1):
    print("Aquí pongo el código a ejecutar.")
    print("Tiene una sangría de 4 espacios.")
```

Estas son las normas más importantes de sintaxis que debemos tener en cuenta para comenzar a programar en Python. Durante el curso, incluiremos algunas otras reglas en la sintaxis o en el estilo según vayamos incluyendo nuevos conceptos.

2.3. Almacenar valores

Unas de las principales herramientas que necesitamos en cualquier lenguaje de programación son elementos que nos permitan almacenar valores para usarlos más adelante en nuestro programa. En este apartado explicaremos cómo se deben declarar las variables y las constantes en Python.

Variables

A la hora de implementar nuestros programas será necesario almacenar algunos valores para consultarlos o modificarlos durante la ejecución. Para poder hacerlo utilizaremos las **variables**. Una variable es un identificador al que le asignaremos un valor y, más adelante, llamando a ese identificador podremos consultar el valor.

En Python la creación de una variable se hace a través de una asignación. Haremos una asignación escribiendo el nombre de un identificador, seguido del símbolo = y el valor que deseamos almacenar en la variable:

```
identificador = [valor]
```

A diferencia de otros lenguajes, no es necesario definir el tipo de dato que almacenará la variable. El motivo es que Python infiere este tipo en el momento de ejecutar la asignación y le asignará el tipo de dato que mejor se adapte al valor que hayamos asignado. Por ejemplo, si asignamos a una variable el valor 'Hola mundo!', esta variable será de tipo cadena de texto. Además, una misma variable puede almacenar diferentes tipos de datos durante la ejecución.

A continuación, veremos algunos ejemplos de variables y de los tipos de datos que se han asignado. Para preguntar qué tipo de dato ha almacenado una variable, usaremos la función **type**:

```
# Ejemplo de variable:  
n = 23  
saludo = 'Hola! Qué tal?'  
type(n) # Devuelve int  
type(saludo) # Devuelve str
```

Constantes

Otro elemento muy utilizado en el desarrollo de programas es el uso de **constantes**. Una constante es un tipo de variable cuyo valor no puede ser modificado durante

toda la ejecución del programa. Se suele utilizar para almacenar valores que necesitaremos tener disponibles a lo largo de la ejecución.

En Python no existe ninguna palabra reservada que nos permita definir una constante. Sin embargo, existen dos posibles soluciones para declarar constantes. Una de ellas es utilizar una variable, escribiendo el identificador en mayúsculas para que sepamos que esa variable no puede ser modificada. Esta solución se refiere a los estilos y tendremos que estar pendientes de que esa constante nunca cambia de valor.

```
# Ejemplo de constantes:
IVA = 0.21

precio = 25
precio_final = precio + (precio * IVA)

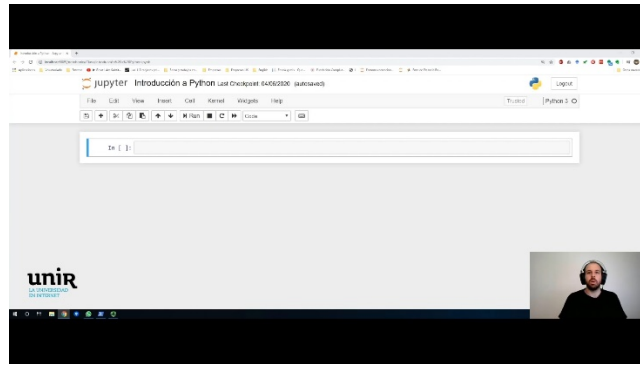
print("El precio final es:", precio_final)
```

Otra solución muy utilizada es crear un *script* de Python, accesible desde nuestro proyecto, donde se almacenen todas las constantes. A continuación, importamos ese *script* con la instrucción `import` para poder llamar a esas constantes. El ejemplo anterior, usando un *script* de constantes, sería el siguiente:

```
# Importamos el fichero de constantes.
import Constantes

precio = 25
precio_final = precio + (precio * Constantes.IVA)

print("El precio final es:", precio_final)
```



Vídeo 1. Sintaxis de Python y almacenar valores.

Accede al vídeo a través del aula virtual

2.4. Tipos de datos y operadores

Como hemos visto, cuando declaramos un elemento para almacenar un valor en Python, ya sea una variable o una constante, no necesitamos definir el tipo de dato que almacenará. Sin embargo, es necesario conocer los tipos de datos que podemos encontrarnos en Python para saber qué operadores podemos aplicar. A continuación, describiremos los tipos de datos básicos que encontraremos en este lenguaje y los operadores que podemos aplicar a cada uno de estos tipos.

Datos numéricos

Los primeros tipos de datos que nos encontramos son los tipos numéricos. Dentro de los tipos numéricos encontramos los siguientes tipos más específicos:

- ▶ **Enteros** (int): 26, 0b1101 (base binaria), 0x3f4a (base hexadecimal).
- ▶ **Flotante** (float): 3.14, 5., -67.763
- ▶ **Complejos** (complex): 0.117j

Usando estos tipos de datos, podemos aplicar los siguientes tipos de operadores: operadores aritméticos, operadores de asignación y operadores de bits. A continuación, revisaremos estas operaciones y su sintaxis en Python.

Operadores aritméticos

Estos operadores incluyen las operaciones matemáticas básicas:

- ▶ **Suma (+):** devuelve como resultado la suma de dos números.

```
3.13 + 6 # Devuelve 9.13
```

- ▶ **Resta (-):** devuelve como resultado la resta de dos números.

```
3.13 - 6 # Devuelve -2.87
```

- ▶ **Multiplicación (*):** devuelve como resultado la multiplicación de dos números.

```
3.13 * 10 # Devuelve 31.3
```

- ▶ **División (/):** devuelve como resultado la división de dos números.

```
3.13 / 10 # Devuelve 0.313
```

- ▶ **División entera (//):** devuelve como resultado la división entera de dos números.

Es decir, el resultado será únicamente la parte entera de la división.

```
3 // 10 # Devuelve 0
```

- ▶ **Módulo (%):** devuelve como resultado el valor del resto obtenido de la división entera entre dos números.

```
3 % 10 # Devuelve 3
```

- ▶ **Exponente (**):** devuelve como resultado el valor exponencial de una base con respecto al exponente:

```
3 ** 2 # Devuelve 9
```

Operadores de asignación

Los operadores de asignación permiten asignar el resultado de la operación a una variable incluyendo el símbolo = en el operador. Estos nos permiten modificar el valor de una variable sin tener que definirla en la parte derecha de la asignación. La lista de operadores de asignación es la siguiente.

- ▶ **Asignación simple (=):** asigna a la variable del lado izquierdo el valor definido en la parte derecha.

```
resultado = 10 # resultado vale 10
```

- ▶ **Suma y asignación (+=):** el operador suma, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado += 10 # resultado vale 20
```

- ▶ **Resta y asignación (-=):** el operador resta, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado -= 10 # resultado vale 0
```

- ▶ **Multipliación y asignación (*=):** el operador multiplica, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado *= 10 # resultado vale 100
```

- ▶ **División y asignación (/=):** el operador divide, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado /= 10 # resultado vale 1
```

- ▶ **División entera y asignación (//=):** el operador realiza la división entera al valor de la variable con respecto al valor definido en el lado derecho:

```
resultado = 14 # resultado vale 14
resultado //= 10 # resultado vale 1
```

- ▶ **Módulo y asignación (%=):** el operador asigna a la variable el resto de la división entera entre el valor de la variable y el valor definido en el lado derecho de la operación:

```
resultado = 14 # resultado vale 14
resultado %= 10 # resultado vale 4
```

- ▶ **Exponente y asignación (**=):** el operador asigna a la variable el resultado del exponente entre el valor de la variable y el valor de la derecha de la operación:

```
resultado = 3 # El resultado vale 3
resultado **= 2 # El resultado vale 9
```

Operaciones de bits

Otra de las operaciones básicas que podemos realizar en Python con valores numéricos son las operaciones de bits. Estas operaciones solo se pueden aplicar a valores enteros. A continuación, repasaremos estas operaciones:

- ▶ **AND (&):** operador lógico *and* a nivel de bits.

```
4 & 5 # El resultado será 4
```

- ▶ **OR (|):** operador lógico *or* a nivel de bits.

```
4 | 5 # El resultado será 5
```

- ▶ **XOR (^):** operador lógico *xor* a nivel de bits.

```
4 ^ 5 # El resultado será 1
```

- ▶ **Mover bits a la izquierda (<<):** operador que mueve todos los bits a la izquierda tantas posiciones como se indique en el lado derecho del operador.

```
4 << 1 # El resultado será 8
```

- ▶ **Mover bits a la derecha (>>):** operador que mueve todos los bits a la derecha tantas posiciones como se indique en el lado derecho del operador.

```
4 >> 1 # El resultado será 2
```

Datos booleanos

El tipo de datos booleano es un tipo binario cuyos valores solo pueden ser True o False. Este tipo de datos es muy utilizado en expresiones de control para diferentes sentencias como son el *if* o el *while*. A este tipo de datos se les pueden aplicar los operadores lógicos:

- ▶ **AND (and):** operador lógico *and*.

```
True and False # Devolverá False
```

- ▶ **OR (or):** operador lógico *or*.

```
True or False # Devolverá True
```

- ▶ **NOT (not):** operador de negación lógica.

```
not True # Devolverá False
```

Además de los operadores lógicos, podemos utilizar los operadores de comparación y de identidad:

- ▶ **Menor (<):** operador que devuelve True si el valor de la izquierda es menor que el valor de la derecha. En caso contrario devolverá False.

```
5 < 7 # Devolverá True
```

- ▶ **Menor o igual (<=):** operador que devuelve True si el valor de la izquierda es menor o igual que el valor de la derecha. En caso contrario devolverá False.

```
7 <= 7 # Devolverá True
```

- ▶ **Mayor (>):** operador que devuelve True si el valor de la izquierda es mayor que el valor de la derecha. En caso contrario devolverá False.

```
5 > 7 # Devolverá False
```

- ▶ **Mayor o igual (>=):** operador que devuelve True si el valor de la izquierda es mayor o igual que el valor de la derecha. En caso contrario devolverá False.

```
7 >= 7 # Devolverá True
```

- ▶ **Igual (==):** operador que devuelve True si el valor de la izquierda es igual que el valor de la derecha. En caso contrario devolverá False.

```
5 == 7 # Devolverá False
```

- ▶ **Distinto (!=):** operador que devuelve True si el valor de la izquierda es distinto que el valor de la derecha. En caso contrario devolverá False.

```
7 != 7 # Devolverá False
```

Cadenas de caracteres

Las cadenas de texto son secuencias de caracteres encapsuladas con comillas simples (") o comillas dobles ("""). A las cadenas de texto almacenadas en variables podemos aplicarle diferentes operadores. Para este apartado crearemos una variable llamada *mensaje* e iremos viendo las diferentes operaciones que podemos aplicar:

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
```

Las cadenas de texto funcionan como una lista de caracteres. Por lo tanto, podemos obtener el carácter que existen en una posición concreta de la cadena. Para ello, aplicamos el operador `[]` indicando la posición a la que queremos acceder dentro de él.

```
mensaje[34] # Devuelve 'e'
```

Este operador no solo nos permite acceder a una única posición, sino que también podemos introducir un rango de caracteres. Para ello usamos este operador separando la posición inicial y la posición final a las que queremos acceder de la siguiente forma:

```
CADENA[POSICION_INICIAL:POSICION_FINAL]
```

Hay que tener en cuenta que esto nos devolverá todos los caracteres desde la `POSICION_INICIAL` hasta la posición anterior a la `POSICION_FINAL`.

```
mensaje[4:15] # Devuelve ' es un mens'
```

Si dejamos la `POSICION_INICIAL` vacía, Python nos devolverá los caracteres desde el comienzo de la cadena hasta la `POSICION_FINAL`. Si, por otro lado, dejamos vacía la `POSICION_FINAL`, nos devolverá los caracteres desde la `POSICION_INICIAL` hasta el final de la cadena de caracteres.

```
mensaje[:20] # Nos devolverá 'Esto es un mensaje d'  
mensaje[40:] # Nos devolverá 'so de Python.'
```

En ambas posiciones se pueden incluir números negativos. En estos casos, Python empezará a contar las posiciones desde el lado derecho.

```
mensaje[-10:] # Desde la posición -10 (es decir, 10 desde la derecha)  
hasta el final. Nos devolverá 'de Python.'  
mensaje[:-10] # Desde la posición inicial, hasta la posición -10 (es  
decir, 10 desde la derecha). Nos devolverá 'Esto es un mensaje de prueba  
para el curso '
```


Operadores de cadenas de texto

Las cadenas de texto cuentan con los operadores que hemos visto en los otros tipos de datos, aunque el resultado está aplicado al dominio de las cadenas de texto. Estos operadores son:

- **Concatenar (+)**: este operador devuelve una cadena de caracteres uniendo los caracteres de dos cadenas.

```
"Hola, " + "estoy bien" # Devolverá 'Hola, estoy bien'
```

- **Multiplicar (*)**: este operador nos permite repetir una cadena de caracteres tantas veces como indiquemos en la parte derecha del operador.

```
"Hoy hace sol! " * 3 # Devolverá 'Hoy hace sol! Hoy hace sol! Hoy hace sol! '
```

Además de poder acceder a posiciones concretas de una cadena de texto y los operadores que hemos visto, Python nos proporciona diferentes funciones para obtener información y manipular cadenas de caracteres. A continuación, veremos las más importantes:

- **len()**: esta función nos permite obtener la longitud de la cadena de caracteres, es decir, nos devuelve cuantos caracteres están contenidos en la cadena.

```
len(mensaje) # Devuelve 53
```

- **find()**: permite obtener la primera posición donde se encuentra la subcadena que pasamos por parámetro dentro de la cadena de texto original. En caso de que la subcadena no exista dentro de la cadena original, nos devolverá -1.

```
mensaje.find('Python') # Devolverá 46
```

```
mensaje.find('Java') # Devolverá -1
```

- **upper()**: convierte todos los caracteres de la cadena de texto en mayúsculas.

```
mensaje.upper() # Devolverá 'ESTO ES UN MENSAJE DE PRUEBA PARA EL CURSO DE PYTHON.'
```

- **lower()**: este operador convierte todos los caracteres de la cadena de texto en minúsculas.

```
mensaje.lower() # Devolverá 'esto es un mensaje de prueba para el curso de python.'
```

- **replace()**: nos permite modificar el contenido de la cadena de caracteres. Para ello, introducimos dos parámetros. El primero de ellos contiene la subcadena que queremos sustituir. El segundo contiene la cadena de texto que sustituirá a la primera subcadena. Si la primera subcadena no existe en la cadena de texto original, no habrá ningún cambio.

```
mensaje.replace("curso", "seminario") # Devolverá 'Esto es un mensaje de prueba para el seminario de Python.'  
mensaje.replace("Java", "C++") # Devolverá 'Esto es un mensaje de prueba para el curso de Python.'
```

Tipo None

El tipo None se utiliza para definir que el valor de una variable es nada o ninguna cosa. Hay que tener cuidado con este tipo de datos, ya que es común utilizarlo cuando queremos declarar una variable, pero no le queremos asignar ningún valor:

```
variable = None  
type(variable) # Devolverá que es de tipo NoneType
```

Es importante saber que None es un tipo de dato propio con su significado y que lo debemos diferenciar de valores por defecto de otros tipos como el booleano (False) o numérico (0). En estos casos None es un tipo y valor diferente.

```
None == False # Devolverá False
```

Transformación de tipos de datos

Por último, una utilidad muy importante en todos los lenguajes de programación es hacer transformaciones entre tipos de datos. A continuación, vamos a hacer un repaso de las transformaciones de tipos de datos permitidas en Python.

- **Convertir a cadena de caracteres** (`str`): convierte un objeto que se pasa por parámetro a cadena de caracteres.

```
valor = 10
str(valor) # Devolverá '10'
```

- **Convertir a valor entero** (`int`): convierte un objeto que se pasa por parámetro al tipo entero. En caso de ser un número decimal, nos devolverá únicamente la parte entera. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
int(valor1) # Devolverá 10
int(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor flotante** (`float`): convierte un objeto que se pasa por parámetro al tipo flotante. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 30
valor2 = 'Hola'
float(valor1) # Devolverá 30.0
float(valor2) # Devolverá un error de tipo ValueError
```

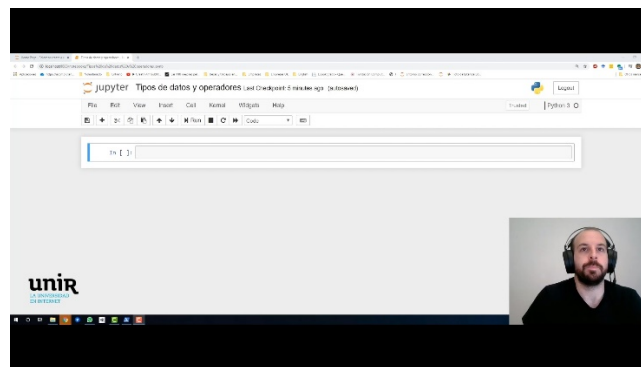
- **Convertir a valor complejo** (`complex`): convierte un objeto que se pasa por parámetro al tipo complejo. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
complex(valor1) # Devolverá 10.78 + 0j
complex(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor booleano** (`bool`): convierte un objeto que se pasa por parámetro al tipo booleano. Su funcionamiento es el siguiente:

- Si no se pasa ningún parámetro, devolverá `False`.
- En el resto de los casos devolverá `True` excepto si: el valor del parámetro es `0`, si es una secuencia vacía de alguna estructura de datos, si se pasa el tipo `None`, si se pasa el valor `False`.

```
valor1 = 10.78
valor2 = False
valor3 = 0
bool(valor1) # Devolverá True
bool(valor2) # Devolverá False
bool(valor3) # Devolverá False
```



Vídeo 2. Tipos de datos y operadores.

Accede al vídeo a través del aula virtual

Introducción a la programación y al análisis de datos
con Python

Programación básica

Índice

Esquema	3
Ideas clave	4
3.1. Introducción y objetivos	4
3.2. Estructuras de datos	5
3.3. Ejecuciones condicionales	13
3.4. Ejecuciones iterativas	16

Programación básica		
Estructuras de datos	Ejecuciones condicionales	Ejecuciones iterativas
Listas	Expresión <i>if</i>	Bucle <i>while</i>
Tuplas	Expresión <i>else</i>	Bucle <i>for</i>
Diccionarios	Expresión <i>elif</i>	Sentencias extras
Conjuntos		Iteradores

3.1. Introducción y objetivos

Este tema introducirá nuevos conceptos que nos permitirán añadir complejidad a nuestros programas. En primer lugar, enumeraremos las estructuras de datos que nos proporciona Python para almacenar objetos más complejos. A continuación, explicaremos las sentencias condicionales que nos permitirán ejecutar diferentes bloques de código dependiendo de situaciones o resultados. Por último, explicaremos las formas que tenemos de ejecutar bloques de código de manera repetida usando los bucles y, además, veremos los iteradores que nos permiten recorrer secuencias de elementos.

Al finalizar este tema, habrás alcanzado los siguientes objetivos:

- ▶ Conocer las diferentes estructuras de datos incluidas en Python.
- ▶ Aprender las funciones más importantes para cada una de las estructuras de datos.
- ▶ Comprender cómo hacer bifurcaciones en la ejecución de los programas con las sentencias condicionales.
- ▶ Conocer las sentencias de los bucles que nos permiten ejecutar bloques de código repetidamente.
- ▶ Saber utilizar los iteradores para recorrer secuencias de valores.

3.2. Estructuras de datos

En el tema anterior vimos los tipos de datos básicos que podemos utilizar en Python para almacenar valores. Sin embargo, este tipo de datos no son suficientes para almacenar valores complejos como, por ejemplo, una lista de usuarios de una página web. En este tema, veremos las estructuras de datos que nos proporciona Python para almacenar valores más complejos.

Listas

Las listas son las estructuras de datos más utilizadas en Python. Una lista es una estructura que nos permite tener un conjunto de objetos separados por comas. Esta estructura de datos es **mutable**, es decir, podemos cambiar el valor de una lista que hemos creado como, por ejemplo, cambiar el orden de los elementos, eliminar elementos, etc.

En una lista pueden existir elementos de diferentes tipos, es decir, elementos con diferentes tipos de datos o, incluso, diferentes estructuras de datos. Sin embargo, lo normal es que todos los elementos de una lista sean del mismo tipo. Para declarar una lista, escribimos entre corchetes ([]) un conjunto de elementos separados por comas:

```
lista = [3, 'Hola', True]
```

También podemos crear una lista vacía usando únicamente los corchetes sin ningún valor dentro o utilizando la función `list()`:

```
lista_vacia = []
```

A partir de una lista que hayamos creado, podemos acceder a los distintos elementos de dicha lista. Para ello, solo tenemos que indicar la posición que ocupa el elemento

dentro de la lista, es decir, indicar el *índice del elemento*. Debemos tener en cuenta que el primer elemento de la lista ocupa la posición 0.

```
lista = [3, 'Hola', True]
lista[2] # Devolverá el valor True
```

También podemos acceder a un conjunto de elementos de una lista usando el símbolo de dos puntos (:) dentro de los corchetes. A esto se le llama un rango de índices. Por ejemplo, para acceder a las 2 primeras posiciones de una lista podríamos hacer lo siguiente:

```
lista = [3, 'Hola', True]
lista[:2] # Devolverá [3, 'Hola']
```

Cuando usamos los rangos de índices, como el visto anteriormente, existen unos valores por defecto cuando no indicamos su valor. El valor por defecto del primer índice es 0 y el valor por defecto del último índice es la longitud de la lista.

```
lista = [3, 'Hola', True]
lista[:2] # Devolverá [3, 'Hola']
lista[1:] # Devolverá ['Hola', True]
```

Además, dentro de los índices de las listas, ya sea un solo índice o un rango, podemos usar valores negativos. En esos casos Python devuelve los elementos contando su posición desde la derecha.

```
lista = [3, 'Hola', True]
lista[-1] # Devolverá True
lista[-2:] # Devolverá ['Hola', True]
```

Funciones aplicables a listas

Cuando usamos listas podemos usar un conjunto de funciones que nos permiten obtener propiedades de las listas o modificarlas. A continuación, vamos a enumerar algunas de las funciones más utilizadas en listas:

- ▶ **len:** devuelve la longitud de una lista, es decir, el número de elementos incluidos en una lista.

```
lista = [3, 'Hola', True]
len(lista) # Devolverá 3
```

- ▶ **index:** devuelve la posición que ocupa un elemento dentro de una lista.

```
lista = [3, 'Hola', True]
lista.index('Hola') # Devolverá 1
```

- ▶ **insert:** inserta un elemento dentro de una lista en la posición que le indicamos.

```
lista = [3, 'Hola', True]
lista.insert(1, 'Adiós')
lista # Nos mostrará [3, 'Adiós', 'Hola', True]
```

- ▶ **append:** inserta un elemento al final de la lista. En el caso de que pongamos una lista de elementos, la función lo insertará como un elemento único.

```
lista = [3, 'Hola', True]
lista.append([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, [3, 4]]
```

- ▶ **extend:** permite agregar un conjunto de elementos en una lista. A diferencia del método anterior, si incluimos una lista de elementos, se agregarán cada uno de los elementos a la lista.

```
lista = [3, 'Hola', True]
lista.extend([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, 3, 4]
```

- ▶ **remove**: elimina el elemento que pasamos por parámetro de la lista. En caso de que este elemento estuviese repetido, solo se eliminará la primera copia.

```
lista = [3, 'Hola', True, 'Hola']
lista.remove('Hola')
lista # Nos mostrará [3, True, 'Hola']
```

- ▶ **count**: devuelve el número de veces que se encuentra un elemento en una lista.

```
lista = [3, 'Hola', True, 'Hola']
lista.count('Hola') # Nos devolverá 2
```

- ▶ **reverse**: este método nos permite invertir la posición de todos los elementos de la lista.

```
lista = [3, 'Hola', True]
lista.reverse()
lista # Nos mostrará [True, 'Hola', 3]
```

- ▶ **sort**: ordena los elementos de una lista. Por defecto, este método los ordena en orden creciente. Para ordenarlo de forma decreciente, hay que incluir el parámetro (`reverse=True`). ¡Ojo! La lista debe contener elementos del mismo tipo.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.sort()
lista # Nos mostrará [0, 1, 4, 5, 6, 7, 9]
lista.sort(reverse=True)
lista # Nos mostrará [9, 7, 6, 5, 4, 1, 0]
```

- ▶ **pop**: elimina y devuelve el elemento que se encuentra en la posición que se pasa por parámetro. En caso de que no se pase ningún valor por parámetro, eliminará y devolverá el último elemento de la lista.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.pop(1) # Devolverá 4
lista # Nos mostrará [6, 1, 9, 7, 0, 5]
```

Tuplas

Al igual que las listas, las tuplas son conjuntos de elementos separados por comas. Sin embargo, a diferencia de las listas, las tuplas son **inmutables**, es decir, no se pueden modificar una vez creadas. Un ejemplo de tupla sería el siguiente:

```
tupla = 'Hola', 3.4, True, 'Hola'
tupla
```

En el ejemplo anterior, hemos creado una tupla a partir de una secuencia de valores separados por comas. El resultado es una tupla que contiene todos estos elementos. A esta operación se le denomina **empaquetado de tuplas**. También disponemos del método inverso. Si a una tupla de longitud n le asignamos n variables, cada una de las variables tendrá uno de los componentes de la tupla. A esta operación se le llama **desempaquetado de tuplas**.

```
w, x, y, z = tupla # w = 'Hola', x = 3.4, y = True, z = 'Hola'
```

Para acceder a los elementos de una tupla, lo haremos de la misma manera que con las listas:

```
tupla[1] # Nos mostrará 3.4
tupla[1:] # Nos mostrará (3.4, True, 'Hola')
```

Funciones aplicables a tuplas

Al igual que las listas, tenemos varios métodos que podemos usar en las tuplas. Los métodos más utilizados son los siguientes:

- **len**: método que devuelve la longitud de la tupla.

```
tupla = 'Hola', 3.4, True, 'Hola'
len(tupla) # Devolverá 4
```

- ▶ **count**: número de veces que se encuentra un elemento en una tupla.

```
tupla.count('Hola') # Devolverá 2
```

- ▶ **index**: devuelve la posición que ocupa un elemento dentro de una tupla. En caso de que el elemento este repetido, devolverá la primera posición donde aparece el objeto.

```
tupla.index('Hola') # Devolverá 0
```

Diccionarios

La última estructura de datos que veremos en Python son los diccionarios. Los diccionarios conforman una estructura que enlaza los elementos almacenados con claves (*keys*) en lugar de índices, como las estructuras anteriores. Es decir, para acceder a un objeto es necesario hacerlo a través de su clave.

La mejor manera de comprender un diccionario es verlo como un conjunto de pares (clave, valor), donde las claves son únicas, es decir, no están repetidas, y nos permiten acceder al objeto almacenado. Para crear un diccionario definiremos un conjunto de elementos clave valor delimitados por llaves ({}):

```
diccionario = {  
    'clave1': 'Mi primer valor',  
    'clave3': 'Y, como no, mi tercer valor',  
    'clave2': 'Este es mi segundo valor'  
}
```

Si queremos acceder a uno de sus valores, necesitaremos conocer su clave y lo pondremos entre corchetes ([]):

```
diccionario['clave1'] # Devolverá 'Mi primer valor'
```

Para crear nuevos elementos en los diccionarios usamos la misma forma de acceso a un elemento, pero asignando un nuevo valor:

```
diccionario['clave_nueva'] = 'nuevo valor'
```

Podemos crear diccionarios vacíos usando las llaves, pero sin insertar ningún elemento, o usando la función `dict()`:

```
diccionario = dict()
```

Podemos eliminar un elemento del diccionario usando la instrucción `del` e indicando qué elemento del diccionario queremos eliminar.

```
del diccionario['clave1'] # Eliminará el elemento con clave 'clave1'
```

Funciones aplicables a diccionarios

A continuación, enumeraremos dos de las funciones más utilizadas en los diccionarios.

- **list**: devuelve una lista de todas las claves incluidas en un diccionario. Si queremos la lista ordenada, usaremos la función **sorted** en lugar de **list**.

```
list(diccionario) # Devolverá ['clave1', 'clave3', 'clave2']
sorted(diccionario) # Devolverá ['clave1', 'clave2', 'clave3']
```

- **in**: comprueba si una clave se encuentra en el diccionario.

```
'clave3' in diccionario # Devolverá True
```

Conjuntos

Los conjuntos son colecciones no ordenadas de elementos. Además, los conjuntos no contienen repetición de elementos, es decir, se eliminan todos los elementos duplicados. Para crear un conjunto podemos indicar un conjunto de objetos separados por comas y delimitados por llaves `{}`.

```
conjunto = {'audi', 'mercedes', 'seat', 'ferrari', 'ferrari', 'renault'}
```

Para crear un conjunto vacío podemos crearlo usando las llaves sin insertar ningún valor o la función `set()`.

```
conjunto = set()  
conjunto # Nos mostrará {}
```

Al no ser una lista ordenada, no podemos acceder a sus elementos a través de su índice; esto nos devolverá un error de tipo.

Funciones aplicables a conjuntos

A continuación, veremos algunas de las operaciones soportadas por los conjuntos de Python.

- ▶ **union:** operación matemática para obtener la unión de dos conjuntos.

```
conjunto1 = {1, 2, 3}  
conjunto2 = {3, 4, 5}  
conjunto1.union(conjunto2) # Devolverá {1, 2, 3, 4, 5}
```

- ▶ **intersection:** operación matemática para obtener la intersección de dos conjuntos.

```
conjunto1 = {1, 2, 3}  
conjunto2 = {3, 4, 5}  
conjunto1.intersection(conjunto2) # Devolverá {3}
```

- ▶ **difference:** operación matemática para obtener la diferencia del conjunto original con respecto al conjunto que se pasa por parámetro, es decir, los elementos del primer conjunto que no están en el segundo.

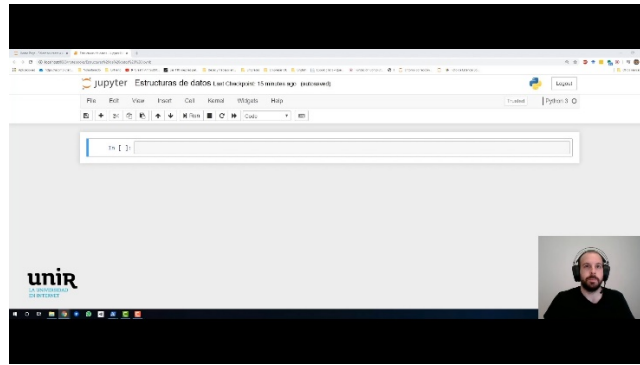
```
conjunto1 = {1, 2, 3}  
conjunto2 = {3, 4, 5}  
conjunto1.difference(conjunto2) # Devolverá {1, 2}
```

- ▶ **in:** comprueba si un elemento se encuentra dentro de un conjunto.

```
1 in conjunto1 # Devolverá True
```

- ▶ **len:** devuelve la longitud del conjunto.

```
len(conjunto1) # Devolverá 3
```

Vídeo 1. Estructuras de datos.

Accede al vídeo a través del aula virtual

3.3. Ejecuciones condicionales

Hasta este momento durante el curso hemos visto algunas sentencias que nos permiten hacer pequeños programas en Python. Sin embargo, en la mayoría de las ocasiones los programas que haremos no tendrán una ejecución lineal, sino que habrá bifurcaciones dependiendo del resultado que obtengamos de algunas evaluaciones. Para poder hacer estas bifurcaciones en la ejecución de nuestro programa es necesario utilizar las ejecuciones condicionales.

Expresión `if`

Las expresiones `if` nos permiten ejecutar un bloque de instrucciones únicamente si la expresión lógica que hemos puesto devuelve `True`. Para escribir una sentencia `if` se sigue la siguiente estructura:

```
if (EXPRESION_LOGICA):  
    sentencia_1  
    sentencia_2  
    ...
```

Aquí tenemos un ejemplo de cómo podríamos usar una sentencia `if`. En este ejemplo pedimos al usuario que escriba un valor entre 1 y 10. A continuación, comprobamos si el número es mayor que 5 y, si es así, imprimimos un mensaje:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
```

Expresión `else`

Por otro lado, si queremos ejecutar otro bloque de instrucciones cuando no se cumple la condición del `if`, usaremos la expresión `else`. Esta expresión debe ir siempre después del bloque de instrucciones del `if`:

```
if (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
else:
    sentencia_1
    sentencia_2
    ...
```

Podemos ampliar el ejemplo anterior para que, en el caso de que el número no sea mayor que 5, mostremos otro mensaje al usuario. Para ello, incluiremos un bloque `else` después del bloque `if`:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
else:
    print("Soy menor o igual que 5")
```

Expresión `elif`

La expresión `elif` nos permite evaluar más condiciones dentro de un `if` para ejecutar otros bloques de instrucciones. Si incluimos un bloque `else`, las instrucciones de este

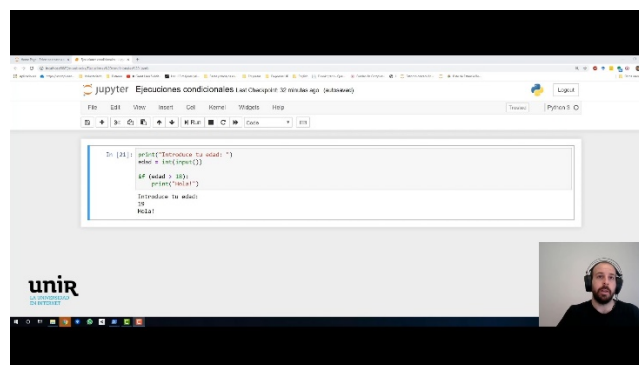
bloque solo si todas las expresiones lógicas de los bloques `if` y `elif` han devuelto `False`. Se puede añadir más de un bloque `elif` y estos bloques deben ir después del bloque `if` y antes del bloque `else`:

```
if (EXPRESION_LOGICA):
    sentencia_1
...
elif (EXPRESION_LOGICA_2):
    sentencia_1
...
elif (EXPRESION_LOGICA_3):
    sentencia_1
...
else:
    sentencia_1
    sentencia_2
...
```

Vamos a aplicar esta expresión en nuestro ejemplo anterior. Para ello, comprobaremos también si el valor introducido por el usuario es 5 y, si es así, mostraremos otro mensaje:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
elif (number == 5):
    print("Soy el número 5")
else:
    print("Soy menor o igual que 5")
```



Vídeo 2. Ejecuciones condicionales.

Accede al vídeo a través del aula virtual

3.4. Ejecuciones iterativas

En esta sección vamos a explicar las dos expresiones que existen en Python para ejecutar varias veces un conjunto de instrucciones. Esas expresiones son la sentencia `while` y la sentencia `for`. Además, veremos otras sentencias que se pueden utilizar en los bucles para modificar el flujo de ejecución y, por último, veremos el uso de iteradores que nos permiten recorrer todos los elementos de objetos como las listas.

Bucle `while`

La primera instrucción que vamos a explicar en las iteraciones es `while`. Esta instrucción repite un bloque de código mientras se cumpla una condición definida por nosotros. Esa condición, al igual que pasaba con `if`, viene dada en forma de expresión lógica. El bloque de instrucciones de la instrucción `while` dejará de ejecutarse cuando esa expresión lógica devuelva un `False`. El formato de escritura de `while` es el siguiente:

```
while (EXPRESION_LOGICA):  
    sentencia_1  
    sentencia_2  
    ...
```

Por ejemplo, imaginemos que queremos imprimir una secuencia de números desde el número 5 hasta el número 0. Esto lo podríamos hacer con un bloque `while`, al cual, mientras el número que estamos imprimiendo sea mayor que 0, le restaremos una unidad y lo imprimiremos:

```
numero = 5  
fin = 0  
  
while(numero > fin):  
    numero -= 1  
    print(numero)
```

A la instrucción `while` se la puede incluir la sentencia `else`. A diferencia de las ejecuciones condicionales, el bloque de instrucciones de la sentencia `else` se

ejecutará siempre cuando acabe de ejecutarse las iteraciones en `while`. En este caso, escribiríamos la sentencia `else` a continuación de las instrucciones del bloque `while`.

```
while (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
else:
    sentencia_1
    sentencia_2
    ...
```

En el ejemplo anterior, podemos imprimir un mensaje al usuario para indicarle que hemos acabado de imprimir números. Para ello, incluimos un bloque `else` después del bloque `while` con la instrucción que imprimirá el mensaje:

```
numero = 5
fin = 0

while(numero > fin):
    numero -= 1
    print(numero)
else:
    print("Ya he acabado!")
```

Esta es la forma más sencilla de crear sentencias iterativas. Normalmente, la instrucción `while` se utiliza para hacer búsquedas de elementos o ejecutar un conjunto de acciones hasta que ocurre un evento. En ambos casos, no se sabe exactamente cuántas ejecuciones tenemos que hacer y depende de la evaluación de una expresión lógica.

Bucle `for`

La segunda forma de crear secuencias iterativas es con la instrucción `for`. Esta permite recorrer un conjunto de elementos (por ejemplo, listas) en cualquier sentido y con cualquier paso. La forma general de crear una sentencia `for` es la siguiente:

```
for VARIABLE in OBJETO:
    sentencia_1
    sentencia_2
    ...
```

```
else:
    sentencia_1
    sentencia_2
    ...
```

En la instrucción `for` declaramos una variable a la que, en cada iteración del bucle, se le asignará el valor de uno de los elementos contenidos en `OBJETO`. Como pasaba en la sentencia `while`, podemos incluir una sentencia `else` que ejecutará sus instrucciones cuando hayan finalizado todas las iteraciones del bucle `for`.

Podemos hacer el mismo ejemplo que en la sentencia `while`. Para ello creamos una lista con los números que queremos imprimir. Con la instrucción `for` recorreremos cada uno de esos números y los imprimimos. Al final, mostraremos un mensaje indicando que hemos acabado:

```
numeros = [5, 4, 3, 2, 1, 0]

for numero in numeros:
    print(numero)
else:
    print("Ya he acabado!")
```

Esta forma de asignación también se puede hacer con cadenas de texto. En este caso, a la variable se le irá asignando cada uno de los caracteres de la cadena en cada paso del bucle:

```
texto = 'Hola mundo'

for caracter in texto:
    print(caracter)
```

Además, como es lógico, la variable puede tener un tipo de dato diferente en cada vuelta del bucle.

```
lista = ['texto', 5, (23, 56)]

for elemento in lista:
    print(elemento)
```

En las sentencias `for`, una forma de recorrer un objeto, como una lista, es a través de sus índices. El objetivo es que la variable tenga como valor en cada paso la posición de la lista que estamos visitando. Para poder hacer esto, en Python se utiliza la instrucción `range`. Esta instrucción nos devuelve un rango de números desde un número inicial hasta uno final, y con la separación entre números que hayamos seleccionado. Su estructura es una de las siguientes:

```
# Secuencia de números de 0 a NUMERO_FINAL con paso 1
range(NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso 1
range(NUMERO_INICIAL, NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso PASO
range(NUMERO_INICIAL, NUMERO_FINAL, PASO)
```

Los rangos se pueden utilizar directamente como objeto en el bucle `for`. Sin embargo, para poder imprimir todos los índices que nos ha generado un rango, es necesario encapsular el rango en una lista.

A continuación, vemos un ejemplo del uso de rangos. En este ejemplo solo queremos mostrar los caracteres que ocupan una posición par en la cadena «Hola mundo», podríamos hacerlo así:

```
cadena = 'Hola mundo'

# Comenzamos en 0, hasta la longitud de la cadena y con paso = 2
for i in range(0, len(cadena), 2):
    print(cadena[i])
```

Sentencias extras

A las estructuras de iteración que hemos visto antes podemos incluir otras sentencias que permiten modificar la ejecución de los bucles. Estas sentencias se pueden utilizar tanto en los bucles `while` como en los bucles `for`.

- **break:** esta sentencia rompe la ejecución del bucle en el momento en que se ejecute.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Rompe el bucle!')
        break

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Rompe el bucle!
```

- **Continue:** esta instrucción permite saltarnos una iteración del bucle sin que se rompa la ejecución final.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Me salto una vuelta')
        continue

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Me salto una vuelta,
6, 7, 8, 9
```

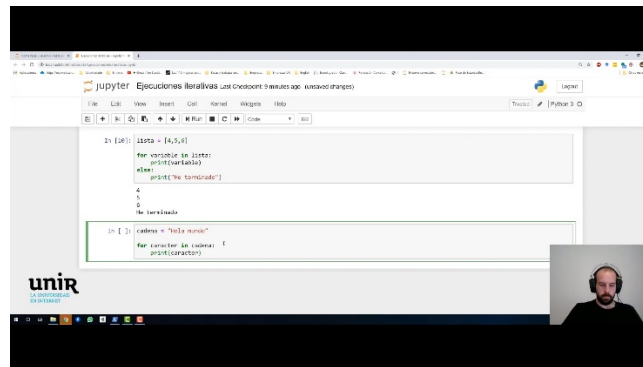
Iteradores

Otra forma de recorrer los elementos de un objeto en Python es utilizando iteradores. Un iterador es un objeto que, al aplicarlo sobre otro objeto iterable, como son las cadenas de texto o los conjuntos, nos permite obtener el siguiente elemento por visitar.

Para crear un iterador sobre un objeto usamos la instrucción `iter(OBJETO)` y se lo asignamos a una variable. Una vez creado, podemos visitar los elementos del OBJETO con la función `next()` y pasando por parámetro el identificador del iterador. Veremos esto en un ejemplo.


```
cadena = 'Hola'
iterador = iter(cadena)

next(iterador) # Devolverá 'H'
next(iterador) # Devolverá 'o'
next(iterador) # Devolverá 'l'
next(iterador) # Devolverá 'a'
```



Vídeo 3. Ejecuciones iterativas.

Accede al vídeo a través del aula virtual

Introducción a la programación y al análisis de datos
con Python

Funciones

Índice

Esquema	3
Ideas clave	4
4.1. Introducción y objetivos	4
4.2. Definición de funciones	4
4.3. Parámetros	6
4.4. Parámetros indeterminados	7
4.5. Retorno de valores	8
4.7. Funciones incluidas en Python	11
4.8. Funciones anónimas	18

Funciones		
Definición de funciones	Librerías estándar	Funciones anónimas
Sentencia <i>def</i>	Librería <i>math</i>	Expresiones <i>lambda</i>
Parámetros	Librería <i>sys</i>	Función <i>filter</i>
Argumentos	Librería <i>os</i>	Función <i>map</i>
Retorno de valores	Librería <i>random</i>	

4.1. Introducción y objetivos

Las funciones tienen dos principales ventajas en la programación. En primer lugar, una función puede implementar una funcionalidad que vamos a utilizar más veces dentro de un programa. En segundo lugar, las funciones nos permiten organizar el código de manera que podamos agrupar bloques de código en funcionalidades concretas. En este tema vamos a explicar cómo implementar las funciones en Python y cómo podemos usarlas. Los objetivos son los siguientes:

- ▶ Aprender cómo definir las funciones.
- ▶ Comprender cómo se usan los parámetros y los argumentos en las funciones.
- ▶ Conocer cómo se devuelven resultados con las funciones.
- ▶ Conocer algunas de las funciones básicas incluidas en Python.
- ▶ Aprender cómo se definen las funciones anónimas en Python.

4.2. Definición de funciones

Las funciones nos permiten encapsular un bloque de instrucciones para que podamos utilizarlo varias veces dentro de nuestros programas. Para hacer esto, es necesario identificar ese bloque de instrucciones usando la sintaxis de funciones que incluye Python. Las funciones se definen con la sentencia `def`. Esta palabra clave indica a Python que vamos a crear un objeto ejecutable que podrá ser llamado más adelante durante la ejecución del programa. El formato para definir una función en Python es:

```
def NOMBRE_FUNCION(ARG1, ARG2,...):  
    sentencia_1  
    sentencia_2  
    ...  
    return OBJETO_DEVOLVER
```

Como podemos observar, la primera línea viene definida por la palabra reservada `def`, seguida del nombre que queramos darle a la función. Este nombre será el identificador que nos permitirá llamar a la función más adelante. A continuación, indicaremos la lista de parámetros que tendrá la función, separados por comas y entre paréntesis. Estos parámetros serán valores u objetos que se necesitarán para ejecutar la función. En el caso de que nuestra función no necesite parámetros, dejaremos los paréntesis vacíos. Al final de la declaración de la función usaremos dos puntos (`:`) para comenzar a escribir las sentencias que se ejecutarán. El bloque de sentencias debe tener una sangría de 4 espacios con respecto a la definición de la función para que el Python entienda que son instrucciones que pertenecen a la función. En el caso de que la función tenga que devolver un valor, usaremos la palabra reservada `return` junto con el identificador o la expresión que devolverá el valor.

Vamos a ver la definición de una función a través de un ejemplo. Imaginemos que queremos crear una función que nos calcule el área de un triángulo. Para poder calcular el área del triángulo, la función necesita dos datos: su base y su altura. Estos dos datos serán los parámetros de la función. Por último, devolveremos el resultado del cálculo usando la sentencia `return`. La función para calcular el área del triángulo quedaría como se muestra a continuación:

```
def area_triangulo(base, altura):  
    area = (base * altura) / 2  
    return area
```

Para ejecutar la función solo debemos usar el nombre de la función e insertar los valores de la base y la altura:

```
area_triangulo(6, 5) # Devolverá 15.0
```

Esta es la forma general de definir una función en Python. Sin embargo, existen muchas opciones que nos permiten definir los parámetros de las funciones.

4.3. Parámetros

En la mayoría de las ocasiones, necesitamos introducir información a las funciones para que puedan realizar la acción que deseamos. Por este motivo, podemos declarar unos parámetros que la función usará para leer esa información. Hay que saber diferenciar entre los **parámetros**, que son los valores que definimos en una función, y los **argumentos**, que son los valores que introducimos a la función en el momento de la ejecución.

En el ejemplo anterior, cuando hemos declarado la función `area_triangulo`, los elementos `base` y `altura` eran los parámetros. Más adelante, cuando hemos indicado que la base era 6 y la altura 5, esos valores son los argumentos de la función.

A la hora de llamar a una función existen dos formas para introducir los argumentos de la función:

- ▶ **Argumentos por posición:** los argumentos se envían en el mismo orden en el que se han definido los parámetros. Esta es la forma que hemos escogido para llamar a la función `area_triangulo` en el ejemplo anterior.
- ▶ **Argumentos por nombre:** los argumentos se envían utilizando los nombres de los parámetros que se han asignado en la función. Para ello, usaremos el nombre del parámetro, seguido del símbolo igual (=) y del argumento. En el ejemplo anterior sería de la siguiente forma:

```
area_triangulo(base=10, altura=4) # Devolverá 20.0
```

Cuando una función tiene definido unos parámetros, es obligatorio que, a la hora de llamarlo, la función reciba el mismo número de argumentos. En caso de que no recibiese alguno de esos argumentos, Python devolvería un error de tipo. Sin embargo, en el momento de declarar una función, podemos asignar un valor por

defecto a cada parámetro. Este valor por defecto se usará solo si no se ha indicado un argumento en el parámetro correspondiente.

Para asignar un valor por defecto a un parámetro, definiremos el nombre del parámetro, seguido por el símbolo = y el valor por defecto que le queramos dar. Por ejemplo, vamos a asignar un valor por defecto para la base y la altura en la función `area_triangulo`.

```
def area_triangulo(base=10, altura=10):  
    area = (base * altura) / 2  
    return area
```

En este caso, si llamásemos a la función `area_triangulo` sin ningún argumento, usaría los valores por defecto que hemos indicado para calcular el área.

```
area_triangulo() # Devolverá 50.0
```

También podemos indicar únicamente uno de los parámetros. En el caso de que lo hagamos por posición, reconocerá que ese argumento es el del parámetro `base`; en cambio, si utilizamos los argumentos por nombre, podemos aplicarlo a cualquiera de los dos parámetros.

Muchas de las funciones que existen en Python tienen parámetros con valores por defecto. Por este motivo, es importante revisar la documentación de una función para saber cómo van a ser los argumentos de la función antes de usarla.

4.4. Parámetros indeterminados

En algunas ocasiones, puede que necesitamos definir una función que necesite un número variable de argumentos. Para estos casos Python nos permite usar los parámetros indeterminados en las funciones. Estos parámetros nos permiten incluir tantos argumentos como queramos en el momento de la ejecución de la función.

Como pasaba con los parámetros normales, existen dos maneras de asignar los argumentos:

- **Argumentos por posición:** se deben definir los parámetros como una lista dinámica. Para ello, a la hora de definir el parámetro, se incluirá un asterisco (*) antes del nombre del parámetro. Los parámetros indeterminados se recibirán por posición. A estos parámetros se les puede pasar cualquier tipo de dato en cada función. Un ejemplo de su uso sería el siguiente:

```
def imprime_numeros(*args):  
    for numero in args:  
        print(numero)  
  
imprime_numeros(1, 7, 89, 46, 9394)
```

- **Argumentos por nombre:** para recibir varios argumentos por nombre, sin saber la cantidad, es necesario definir los parámetros como un diccionario dinámico. Para ello se usa dos asteriscos (**) antes del nombre del parámetro. Un ejemplo de su uso sería el siguiente:

```
def imprime_valores(**args):  
    for argumento in args:  
        print(argumento, '=>', args[argumento])  
  
imprime_valores(arg1='Hola', arg2=[2,3,4], arg3=876.98)
```

A la hora de declarar una función, podemos combinar las dos formas de declarar parámetros normales con las formas que hemos visto para declarar parámetros indeterminados.

4.5. Retorno de valores

Uno de los objetivos más comunes en las funciones es que realicen una operación y devuelvan el resultado de la misma. Para devolver uno o varios valores se utiliza la sentencia `return`. En el siguiente ejemplo tenemos una función que devuelve la potencia entre dos números.

```
def potencia(base, exponente):  
    return base ** exponente
```

Como vemos, la única instrucción que tenemos dentro de la función es una sentencia `return` que devuelve el resultado de la operación. Hay que tener en cuenta que la instrucción `return` debe ser la última instrucción para ejecutarse, ya que en ese momento Python saldrá de la función.

En Python, las funciones pueden devolver más de un valor a la vez. Para hacer esto, solo tenemos que separar por comas todos los valores que queramos devolver después de la sentencia `return`. En el siguiente ejemplo, la función devuelve tres objetos de diferentes tipos:

```
def ejemplo():  
    return "Hola", 3546, [3,90]
```

Cuando devolvemos más de un valor en una función, lo que obtenemos es una tupla con todos los valores. Por este motivo, si queremos que cada uno de los resultados esté en una variable distinta, es necesario hacer un desempaquetado de la tupla. En el ejemplo anterior sería así:

```
var1, var2, var3 = ejemplo() # Nos dará var1 = "Hola", var2 = 3546, var3 =  
[3, 90]
```

4.6. Documentar funciones

Cuando estamos desarrollando *software*, es muy importante documentar bien las secciones de código que vamos implementando. Esto nos permitirá recordar qué hace el código que implementamos o muestra a otros desarrolladores lo que queremos hacer. La documentación cobra mayor importancia cuando se trata de funciones. Las funciones encapsulan un comportamiento dentro de un identificador

y otros usuarios no deberían acceder al código de una función para saber qué es lo que se quiere hacer.

Para poder documentar las funciones se utilizan los docstring. En Python todos los objetos cuentan con una variable por defecto llamada `doc`, y nos permite acceder a la documentación de dicho objeto. Para documentar una función usando los docstring, únicamente tenemos que incluir un comentario inmediatamente después de la cabecera de la función. A continuación, se muestra cómo podríamos documentar la función `potencia` que hemos definido antes:

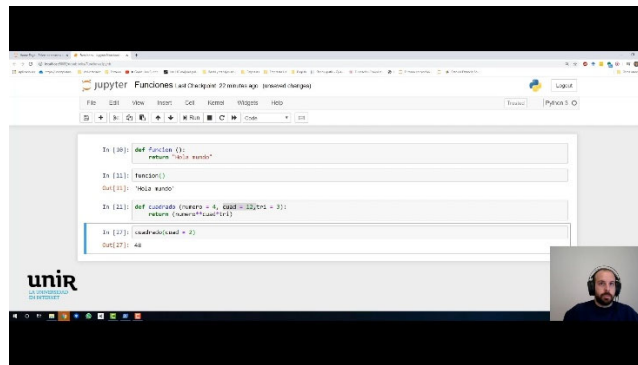
```
def potencia(base, exponente):  
    """  
    Función que calcula la potencia de dos números.  
  
    Argumentos:  
    base -- base de la operación.  
    exponente -- exponente de la operación.  
    """  
    return base ** exponente
```

Al documentar de esta forma las funciones, podemos usar la sentencia `help()` con el nombre de la función para saber la documentación que tiene. En el ejemplo anterior se vería del siguiente modo:

```
help(potencia)  
  
Help on function potencia in module __main__:  
  
potencia(base, exponente)  
    Función que calcula la potencia de dos números.  
  
    Argumentos:  
    base -- base de la operación.  
    exponente -- exponente de la operación.
```

Figura 1. Sentencia `help()`.

En la guía de estilos oficial de Python (PEP8), hay varias reglas y consejos para documentar correctamente nuestro código usando los docstring.



Vídeo 1. Funciones.

Accede al vídeo a través del aula virtual

4.7. Funciones incluidas en Python

Python cuenta con bastantes módulos que incluyen funciones muy útiles para desarrollar nuestros programas. A continuación, veremos los módulos y las funciones más importantes.

Módulo `math`

El módulo `math` es un módulo matemático que incluye numerosas funciones matemáticas que nos pueden ser útiles en el desarrollo de nuestros proyectos. Para utilizar estas funciones es necesario importar el módulo `math` al principio de nuestro código usando la sentencia `import`:

```
import math
```

A continuación, enumeraremos las funciones más útiles agrupadas por tipos.

Funciones aritméticas

Estas funciones nos permiten realizar varias operaciones aritméticas como calcular los valores superiores o inferiores de un número, cálculos factoriales o el máximo común divisor. Las funciones más importantes son:

- ▶ **fabs(x)**: esta función devuelve el valor absoluto del valor x.

```
math.fabs(-1234) # Devolverá 1234
```

- ▶ **gcd(x,y)**: esta función devuelve el máximo común divisor de dos valores x e y.

```
math.gcd(34, 82) # Devolverá 2
```

- ▶ **floor(x)**: esta función devuelve el valor entero más grande que sea menor o igual a x.

```
math.floor(245.89) # Devolverá 245
```

- ▶ **ceil(x)**: esta función devuelve el valor entero más pequeño que sea mayor o igual a x.

```
math.ceil(245.89) # Devolverá 246
```

- ▶ **factorial(x)**: esta función calcula el factorial del número x.

```
math.factorial(5) # Devolverá 120
```

- ▶ **trunc(x)**: esta función devuelve la parte entera del número x.

```
math.trunc(5.7836) # Devolverá 5
```

Funciones trigonométricas

Estas funciones nos permiten hacer cálculos trigonométricos que relacionan los lados de los triángulos con sus ángulos. En todas estas funciones debemos tener en cuenta que se trabaja con los ángulos en radianes. Las funciones más importantes son:

- ▶ **sin(x)**: devuelve el valor del seno del ángulo x en radianes.

```
math.sin(math.pi/4) # Devolverá 0.7071067811865476
```

- ▶ **cos(x)**: devuelve el valor del coseno del ángulo x en radianes.

```
math.cos(math.pi) # Devolverá -1.0
```

- ▶ **tan(x)**: esta función devuelve el valor de la tangente del ángulo x en radianes.

```
math.tan(math.pi/2) # Devolverá 1.633123935319537e+16
```

- ▶ **asin(x)**: esta función calcula el valor del ángulo para que su seno sea x.

```
math.asin(1) # Devolverá 1.5707963267948966
```

- ▶ **acos(x)**: esta función calcula el valor del ángulo para que su coseno sea x.

```
math.acos(1) # Devolverá 0.0
```

- ▶ **atan(x)**: esta función calcula el valor del ángulo para que su tangente sea x.

```
math.atan(1) # Devolverá 0.7853981633974483
```

- ▶ **hypot(x, y)**: esta función calcula la longitud de la hipotenusa de un triángulo rectángulo a partir de los valores de los dos catetos (x, y).

```
math.hypot(10, 7) # Devolverá 12.206555615733702
```

Funciones exponenciales y logarítmicas

Este conjunto de funciones nos permite hacer cálculos sencillos de valores logarítmicos o exponenciales. Las funciones más importantes son:

- ▶ **log(x, [base])**: esta función permite calcular el logaritmo de x. Se puede especificar la base del algoritmo, aunque este argumento no es obligatorio y por defecto se calcula sobre base e.

```
math.log(148.41315910257657) # Devolverá 5.0
```

```
math.log(148.41315910257657, 2) # Devolverá 7.213475204444817
```

```
math.log(148.41315910257657, 10) # Devolverá 2.171472409516258
```

- ▶ **log2(x)**: esta función permite calcular el logaritmo de x con base 2. Devuelve un resultado más preciso que usando la función `log(x, 2)`.

```
math.log2(148.41315910257657) # Devolverá 7.2134752044448165
```

- ▶ **log10(x)**: esta función permite calcular el logaritmo de x con base 10. Devuelve un resultado más preciso que usando la función `log(x, 10)`.

```
math.log10(148.41315910257657) # Devolverá 2.171472409516259
```

- ▶ **pow(x, y)**: esta función permite calcular el valor de x elevado a la potencia y.

```
math.pow(2, 3) # Devolverá 8
```

- ▶ **sqrt(x)**: esta función calcula la raíz cuadrada del valor x.

```
math.sqrt(256) # Devolverá 16
```

Además, el módulo `math` cuenta con dos valores constantes que nos pueden ser de utilidad:

- ▶ **pi**: contiene el valor del número pi.

```
math.pi # Devolverá 3.141592653589793
```

- ▶ **e**: contiene el valor del número e.

```
math.e # Devolverá 2.718281828459045
```

Módulo `sys`

Este módulo proporciona variables y métodos relacionados directamente con el intérprete de Python. En primer lugar, veremos algunas de las variables más destacadas de este módulo:

- ▶ **argv**: devuelve una lista con todos los argumentos que se han pasado por línea de comandos al ejecutar el *script*. Por ejemplo, si ejecutásemos la siguiente instrucción en nuestra consola de comandos:

```
python test.py Hola 25
```

Podríamos obtener los argumentos de la siguiente manera:

```
sys.argv # Devolverá ['test.py', 'Hola', 25]
```

- ▶ **executable:** devuelve la ruta absoluta del fichero ejecutable del intérprete de Python.

```
sys.executable # Devolverá '/usr/local/Python/bin/python3.7'
```

- ▶ **version:** devuelve la versión de Python que se está ejecutando.

```
sys.version # Devolverá '3.7.8'
```

- ▶ **platform:** devuelve la plataforma sobre la que se está ejecutando Python.

```
sys.platform # Devolverá 'darwin'
```

Algunos de los métodos más destacados del módulo `sys` son los siguientes:

- ▶ **exit():** termina la ejecución del intérprete de Python.

```
sys.exit() # Cerrará el intérprete de Python
```

- ▶ **getdefaultencoding():** devuelve el sistema de codificación por defecto del sistema.

```
sys.getdefaultencoding() # Devolverá 'utf-8'
```

Módulo `os`

Este módulo proporciona acceso a variables y funciones que interactúan directamente con el sistema operativo. Este módulo está incluido siempre en las distribuciones de Python. A continuación, veremos los elementos más importantes.

Métodos para archivos y directorios

Estos métodos nos permiten trabajar con las funcionalidades del sistema operativo para crear, eliminar o modificar archivos y directorios. Los métodos más destacados son:

- ▶ **getcwd():** devuelve la ruta del directorio en el que nos encontramos.

```
os.getcwd() # Devolverá /User/UNIR/Python (por ejemplo)
```


- ▶ **mkdir(path)**: crea un nuevo directorio en la ruta que se ha especificado en el argumento.

```
os.mkdir('/NuevaCarpeta') # Creará una carpeta en la dirección  
/NuevaCarpeta
```

- ▶ **rmdir(path)**: elimina el directorio de la ruta que se ha especificado en el argumento.

```
os.rmdir('/NuevaCarpeta') # Eliminará la carpeta /NuevaCarpeta
```

- ▶ **remove(path)**: elimina el fichero de la ruta que se ha especificado en el argumento.

```
os.remove('/NuevaCarpeta/fichero.txt') # Eliminará el archivo  
/NuevaCarpeta/fichero.txt
```

- ▶ **rename(name1, name2)**: renombra el fichero con nombre name1 y los sustituye por name2.

```
os.rename('/NuevaCarpeta/fichero.txt', '/NuevaCarpeta/archivo.txt') #  
Modificará el nombre de fichero.txt por archivo.txt
```

El módulo `os` incluye otro módulo llamado `path` que nos permite acceder a métodos asociados a los nombres de los ficheros y sus rutas. Para ello tenemos que importar el módulo `os.path`.

```
import os.path
```

Los métodos más importantes son:

- ▶ **abspath(ruta)**: devuelve la ruta absoluta de un fichero.

```
os.path.abspath('./fichero.txt') # Devolverá  
/NuevaCarpeta/fichero.txt
```

- ▶ **basename(ruta)**: devuelve el último componente de la ruta que se pasa por parámetro.

```
os.path.basename('/NuevaCarpeta/fichero.txt') # Devolverá fichero.txt
```

- ▶ **exists(ruta):** comprueba si un directorio existe en la ruta especificada.

```
os.path.exists('/NuevaCarpeta') # Devolverá True si existe
```

- ▶ **isfile(ruta):** comprueba si la ruta especificada corresponde a un fichero.

```
os.path.isfile('/NuevaCarpeta/fichero.txt') # Devolverá True
```

- ▶ **isdir(ruta):** comprueba si la ruta especificada corresponde a un directorio.

```
os.path.isdir('/NuevaCarpeta') # Devolverá True
```

Módulo `random`

Este módulo nos proporciona métodos para obtener valores aleatorios. Entre los métodos que destacamos se encuentran los siguientes:

- ▶ **randint(x, y):** devuelve un número aleatorio entre x e y.

```
random.randint(1,10) # Devolverá, por ejemplo, 7
```

- ▶ **choice(secuencia):** devuelve un dato aleatorio de los datos de la secuencia.

```
random.choice(['Hola', 3, [2, 3], True]) # Devolverá, por ejemplo,  
True
```

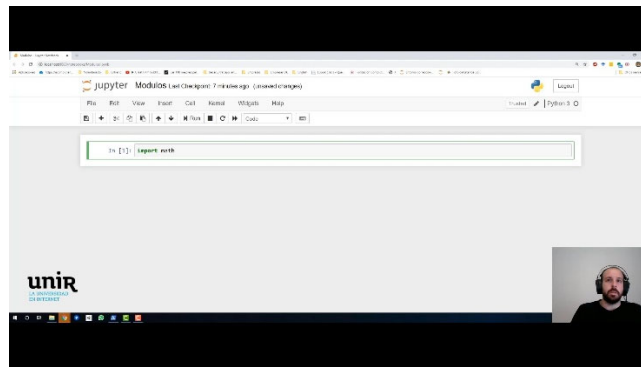
- ▶ **shuffle(secuencia):** permuta los elementos de una secuencia de forma aleatoria.

```
lista = ['Hola', 3, [2, 3], True]  
random.shuffle(lista)  
lista # Mostrará, por ejemplo, [3, 'Hola', True, [2, 3]]
```

- ▶ **sample(secuencia, n):** devuelve n elementos aleatorios de una secuencia.

```
lista = ['Hola', 3, [2, 3], True]  
random.sample(lista, 2) # Devolverá, por ejemplo, [[2, 3], True]
```

Además de todos estos módulos, existen muchos más que nos permiten hacer conexiones a Internet, cortar la longitud de un texto, etc. Todos estos módulos se pueden consultar desde la documentación oficial de Python.



Vídeo 2. Funciones incluidas en Python.

Accede al vídeo a través del aula virtual

4.8. Funciones anónimas

Las funciones anónimas son funciones a las cuales no les vamos a asignar un identificador para ejecutarlas. Es decir, no usaremos la cabecera `def NOMBRE_FUNCION` para definir las. El objetivo de estas funciones es el mismo que el de las funciones normales, con la salvedad de que en estas funciones no podemos incluir un bloque de código, solo una única expresión.

Para implementar las funciones anónimas en Python, usaremos las expresiones `lambda`. Estas expresiones son muy potentes, aunque algo confusas, sobre todo cuando se empiezan a utilizar.

Para explicar una función `lambda` usaremos un ejemplo. Imaginemos que tenemos una función normal que recibe un número y devolvemos su valor al cuadrado. Esta función se definiría del siguiente modo:

```
def cuadrado(x):  
    resultado = x ** 2  
    return resultado
```

Podríamos ejecutar esta función y vemos que nos devolverá el cuadrado del número que hemos introducido:

```
cuadrado(8) # Devolverá 16
```

Para convertir una función normal como esta en una función anónima, necesitamos reducirlo a una única expresión. En este ejemplo, se puede ver claramente que la expresión que calcula el cuadrado es `x ** 2`. Con esta expresión podríamos convertir la función en una expresión `lambda`. Para ello seguimos este esquema:

```
lambda parámetro_1, parámetro_2: expresión
```

En primer lugar, utilizamos la palabra reservada «`lambda`» seguida de los parámetros que tendrá la función, todos ellos separados por comas. A continuación, ponemos dos puntos (:) y la expresión que queremos evaluar. Para nuestro ejemplo de calcular el cuadrado de un número, su expresión `lambda` sería la siguiente:

```
lambda x: x ** 2
```

Esta expresión nos devuelve un objeto función, el cual podemos asignar a una variable y utilizarlo más adelante:

```
cuadrado = lambda x: x ** 2  
cuadrado(8) # Devolverá 16
```

La principal ventaja de utilizar este tipo de funciones es combinándolo con las funciones `filter()` o `map()`. La función `filter` nos permite que filtremos elementos de una secuencia si cumplen una función condicional. Esta función debe recibir dos argumentos, el primero de ellos, una función y el segundo, un objeto de tipo secuencia. Vamos a ver un ejemplo en el que buscamos los números pares de una lista.

```
# Definimos la lista con los números que vamos a analizar.
lista = [1,2,3,4,5,6,7,8,9,10]

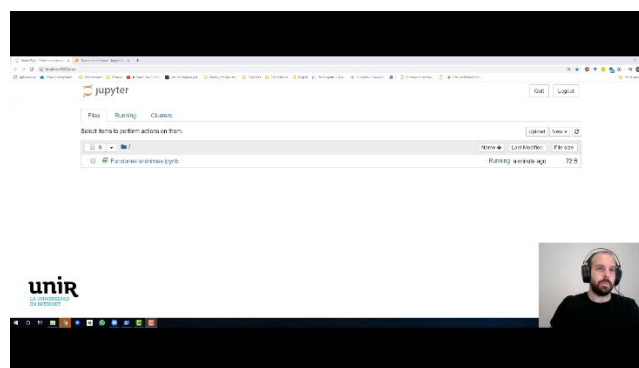
# Definimos la expresión lambda que analiza si un número es par o no.
es_par = lambda numero: numero % 2 == 0

# Aplicamos el filtro para obtener los números pares. Tenemos que insertarlo
en una lista para tener el resultado en formato lista.
list(filter(es_par, lista)) # Nos devolverá [2, 4, 6, 8, 10]
```

La función `map` nos permite aplicar una función a todos los elementos de una secuencia. Para ello, pasamos como argumentos de la función `map`, en primer lugar, la función que queremos aplicar y, luego, el objeto con los elementos a los que se les quiere aplicar la función. Por ejemplo, vamos a aplicar la función `cuadrado` a la lista anterior:

```
# Definimos la lista con los números que vamos a analizar.
lista = [1,2,3,4,5,6,7,8,9,10]

# Aplicamos el filtro para obtener los números pares. Tenemos que insertarlo
en una lista para tener el resultado en formato lista.
list(map(cuadrado, lista)) # Nos devolverá [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



Vídeo 3. Funciones anónimas.

Accede al vídeo a través del aula virtual
